TESI DOCTORAL

Títol **From cluster databases to cloud storage: Providing transactional support on the cloud.**

Realitzada per **Joan Navarro Martín**

en el Centre **Escola Tècnica Superior d'Enginyeria Electrònica i Informàtica La Salle**

i en el Departament d'**Enginyeria.**

Dirigida per **Dr. José Enrique Armendáriz Íñigo** i **Dr. August Climent Ferrer.**

C. Claravall, 1-3
08022 Barcelona
Tel. 936 022 200
Fax 936 022 249
E-mail: urlsc@sec.url.es
www.url.es

Aquesta Tesi Doctoral ha estat defensada el dia ____ d __________________ de ____

al Centre **Escola Tècnica Superior d'Enginyeria Electrònica i Informàtica La Salle**

de la Universitat Ramon Llull

davant el Tribunal format pels Doctors sotasignants, havent obtingut la qualificació:

President/a

____

Vocal

____

Secretari/ària

____

Doctorand/a

Joan Navarro Martín

FROM CLUSTER DATABASES TO CLOUD STORAGE: PROVIDING TRANSACTIONAL SUPPORT ON THE CLOUD

by
JOAN NAVARRO MARTÍN
Grup de Recerca en Internet Technologies & Storage
Enginyeria i Arquitectura La Salle
Universitat Ramon Llull

Submitted in accordance with the requirements for the Degree of
Doctor of Philosophy

Supervised by:

Dr. José Enrique Armendáriz Íñigo

Dr. August Climent Ferrer

Dedicated to my parents, Juan Ramón and Gemma.

ABSTRACT

Over the past three decades, technology constraints (e.g., capacity of storage devices, communication networks bandwidth) and an ever-increasing set of user demands (e.g., information structures, data volumes) have driven the evolution of distributed databases. Since flat-file data repositories developed in the early eighties, there have been important advances in concurrency control algorithms, replication protocols, and transactions management. However, modern concerns in data storage posed by Big Data and cloud computing—related to overcome the scalability and elasticity limitations of classic databases—are pushing practitioners to relax some important properties featured by transactions, which excludes several applications that are unable to fit in this strategy due to their intrinsic transactional nature.

The purpose of this thesis is to address two important challenges still latent in distributed databases: (1) the scalability limitations of transactional databases and (2) providing transactional support on cloud-based storage repositories. Analyzing the traditional concurrency control and replication techniques, used by classic databases to support transactions, is critical to identify the reasons that make these systems degrade their throughput when the number of nodes and/or amount of data rockets. Besides, this analysis is devoted to justify the design rationale behind cloud repositories in which transactions have been generally neglected. Furthermore, enabling applications which are strongly dependent on transactions to take advantage of the cloud storage paradigm is crucial for their adaptation to current data demands and business models.

This dissertation starts by proposing a custom protocol simulator for static distributed databases, which serves as a basis for revising and comparing the performance of existing concurrency control protocols and replication techniques. As this thesis is especially concerned with transactions, the effects on the database scalability of different transaction profiles under different conditions are studied. This analysis is followed by a review of existing cloud storage repositories—that claim to be highly dynamic, scalable, and available—, which leads to an evaluation of the parameters and features that these systems have sacrificed in order to meet current large-scale data storage demands.

To further explore the possibilities of the cloud computing paradigm in a real-world scenario, a cloud-inspired approach to store data from Smart Grids is presented. More specifically, the proposed architecture combines classic database replication techniques and epidemic updates propagation with the design principles of cloud-based storage. The key insights collected when prototyping the replication and concurrency control protocols at the database simulator, together with the experiences derived from building a large-scale storage repository for Smart Grids, are wrapped up into what we have coined as Epidemia: a storage infrastructure conceived to provide transactional support on the cloud. In addition to inheriting the benefits of highly-scalable cloud repositories, Epidemia includes a transaction management layer that forwards client transactions to a hierarchical set of data partitions, which allows the system to offer different consistency levels and elastically adapt its configuration to incoming workloads.

Finally, experimental results highlight the feasibility of our contribution and encourage practitioners to further research in this area.

CURRICULUM VITAE

Education

- **PhD in Communication and Information Technologies and their Management:** 2008 – present. La Salle, Universitat Ramon Llull. *Dissertation title:* "From static to dynamic distributed databases: Providing transactional support on the cloud".
- **Diploma of Advanced Studies:** 2008 – 2010. La Salle, Universitat Ramon Llull. *Dissertation title:* "Optimistic concurrency control in partially replicated distributed databases".
- **MS in Telecommunication Engineering:** 2006 – 2008. La Salle, Universitat Ramon Llull. *Thesis title:* "Optimistic concurrency control in distributed databases".
- **BS in Telecommunication Engineering majoring in Telematics:** 2003 – 2006. La Salle, Universitat Ramon Llull. *Thesis title:* "Practical sessions design of the subject Digital Circuits".

Participation in research projects

- **SHE CLOUD.** Smart Hybrid Enterprise Cloud: 2014 – present.
- **PATRICIA.** Pain and Anxiety Treatment based on social Robot Interaction with Children to Improve pAtient experience: 2013 – present.
- **SPE.** Social Proximity Enabler (Telefonica I+D): 2011 – 2012.
- **INTEGRIS.** INTelligent Electrical GRId Sensor communications: 2009 – 2013.
- **ABIQUO.** ABIQUO the open source cloud company: 2009 – 2011.
- **CONDEP.** From Static to Dynamic Environments: Consistency, Recovery and Dependability Issues: 2006 – 2009.

Work experience

- **Member** of the Grup de Recerca en Internet Technologies & Storage (GRITS), La Salle, Universitat Ramon Llull, (2014 – present).
- **Research visitor** of the Human Sensing Lab, Carnegie Mellon University, (Summer, 2012, 2013, 2014).
- **Research visitor** of the Distributed Systems Group at UPNa (GSD), Universidad Pública de Navarra, (2010, 2011, 2012, 2013).
- **Associate lecturer** at La Salle, Universitat Ramon Llull, (2010 – present).
- **Member** of the Research Group in Distributed Systems and Telematics (GRSDT), La Salle, Universitat Ramon Llull, (2007 – 2014).

Patents

- María José Tomé, David Llanos, **Joan Navarro**. *Method for estimating proximity between users of communication services.* Patent status: Pending (European patent application number EP11382403.1).

Journal papers

- Itziar Arrieta-Salinas, José Enrique Armendáriz-Íñigo, **Joan Navarro**. *Epidemia: Variable consistency for transactional cloud databases.* Journal of Universal Computer Science, 2015 *(in press)*.

- Andreu Sancho-Asensio, **Joan Navarro**, Itziar Arrieta-Salinas, José Enrique Armendáriz-Íñigo, Virginia Jiménez-Ruano, Agustín Zaballos, Elisabet Golobardes. *Improving data partition schemes in Smart Grids via clustering data streams.* Expert Systems with Applications, Volume 41, Issue 13, pp. 5832-5842. 2014. ISSN: 0957-4174.

- David Vernet, Xavi Canaleta, Gemma Pallàs, **Joan Navarro**, Agustín Zaballos. *Setting up and tutoring the working groups of a virtual learning community.* Journal of Research and Practice in Information Technology, 2013 *(in press)*.

- Josep Maria Selga, Agustín Zaballos, **Joan Navarro**. *Solutions to the computer networking challenges of the distribution Smart Grid.* IEEE Communications Letters, Volume 17, Issue 3, pp. 588-591. February 2013. ISSN: 1089-7798.

- **Joan Navarro**, Agustín Zaballos, Andreu Sancho-Asensio, Guillermo Ravera, José Enrique Armendáriz-Íñigo. *The information system of INTEGRIS: INTelligent Electrical GRId Sensor communications.* IEEE Transactions on Industrial Informatics, Volume 9, Issue 3, pp. 1548-1560. November, 2012. ISSN: 1551-3203.

- **Joan Navarro**, José Enrique Armendáriz-Íñigo, August Climent. *An adaptive and scalable replication protocol on power Smart Grids.* Journal in Scalable Computing: Practice and Experience, Volume 12, Number 3, pp. 353-364. September, 2011. ISSN: 1895-1767.

Conference papers

- Aitor Jiménez-Yáñez, **Joan Navarro**, Francesc Muñoz-Escoí, Itziar Arrieta-Salinas, José Enrique Armendáriz-Íñigo. *Chive: A simulation tool for epidemic data replication protocols benchmarking.* The 9th International Conference on Software Engineering and Applications (ICSOFT-EA 2014). August 29-31, 2014, Vienna, Austria. Conference Proceedings. INSTICC Press.

- Xavi Canaleta, **Joan Navarro**, Xavi Solé, David Vernet, Pau López. *Método no formal para la evaluación de la docencia aplicada al Grado de Ingeniería Informática.* Actas de las XX Jornadas de la Enseñanza Universitaria de la Informática (JENUI 2014), July 9-11, 2014, Oviedo, Spain. Conference proceedings.

- **Joan Navarro**, Xavi Canaleta, David Vernet, Xavi Solé, Virginia Jiménez-Ruano, Núria Costa. *Motivación, desmotivación, sobremotivación, y daños colaterales.* Actas de las XX Jornadas de la Enseñanza Universitaria de la Informática (JENUI 2014), July 9-11, 2014, Oviedo, Spain. Conference proceedings.

- Xavi Solé, **Joan Navarro**, Andreu Sancho-Asensio, Agustín Zaballos, Virginia Jiménez-Ruano, Xavi Canaleta, David Vernet. *Sagittarius: Una herramienta para potenciar el trabajo colaborativo en entornos virtuales de aprendizaje.* CISTI'2014 - 9ª Conferencia Ibérica de Sistemas y Tecnologías de Información, June 18-21, 2014, Barcelona, Spain. Conference proceedings.

- Andreu Sancho-Asensio, Xavi Solé, José Antonio Montero, **Joan Navarro**, Xavi Canaleta, David Vernet. *Herramienta de soporte para la formación de grupos de trabajo en entornos de aprendizaje colaborativo.* CISTI'2014 - 9ª Conferencia Ibérica de Sistemas y Tecnologías de Información, June 18-21, 2014, Barcelona, Spain. Conference proceedings.

- **Joan Navarro**, Andreu Sancho-Asensio, Agustín Zaballos, Virginia Jiménez-Ruano, David Vernet, José Enrique Armendáriz-Íñigo. *The management system of INTEGRIS: Extending the Smart Grid to the Web of Energy.* The 4th International Conference on Cloud Computing and Services Science (CLOSER 2014), April 3-5, 2014, Barcelona, Spain. Conference Proceedings, INSTICC Press.

- **Joan Navarro**, Andreu Sancho-Asensio, Carles Garriga, Jordi Albo-Canals, Julio Ortiz-Villajos Maroto, Cristobal Raya, Cecilio Angulo, David Miralles. *A Cloud Robotics Architecture to Foster Individual Child Partnership in Medical Facilities*. Cloud Robotics Workshop in 26th IEEE/RSJ International Conference on Intelligent Robots and Systems. November 3-8, 2013, Tokyo Big Sight, Japan. Conference proceedings.

- Andreu Sancho-Asensio, **Joan Navarro**, Itziar Arrieta-Salinas, José Enrique Armendáriz-Íñigo, Elisabet Golobardes. *Data replication in Smart Grids: Using genetic algorithms to build online partitioning schemes*. 19th International Conference on Soft Computing MENDEL (MENDEL 2013). June 26-28, 2013, Brno, Czech Republic. Conference Proceedings.

- Xavi Canaleta, David Vernet, **Joan Navarro**. *Metodología on demand para el desarrollo de la asignatura de sistemas operativos avanzados.* Actas de las XIX Jornadas de la Enseñanza Universitaria de la Informática (JENUI 2013), Castelló, Spain. Conference proceedings.

- Mariela Louis-Rodríguez, **Joan Navarro**, Itziar Arrieta-Salinas, Ainhoa Azqueta-Alzuaz, Andreu Sancho-Asensio, José Enrique Armendáriz-Íñigo. *Workload management for dynamic partitioning schemes in replicated databases.* The 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013), May 8-10, 2013, Aachen, Germany. Conference Proceedings, INSTICC Press.

- **Joan Navarro**, Xavi Canaleta, Xavi Solé, Marta Arce-Urriza, José Enrique Armendáriz-Íñigo. *A critical approach to modern learning methods.* XIV International Symposium on Educative Computing (SIIE12), October 29 - 31, 2012, Andorra. Conference Proceedings, IEEE Computer Society.

- Itziar Arrieta-Salinas, **Joan Navarro**, José Enrique Armendáriz-Íñigo. *Classic database replication techniques on the cloud.* The 7th International Conference on Availability, Reliability and Security (AReS 2012), August 20-24, 2012, Prague, Czech Republic. Conference Proceedings, IEEE Computer Society.

- Xavi Canaleta, Xavi Solé, **Joan Navarro**. *Herramienta de soporte a la evaluación del aprendizaje y gestión docente.*, XVIII Jornadas de Enseñanza Universitaria de la Informática (JENUI 2012). July 10-13, 2012, Ciudad Real, Spain. Conference proceedings.

- **Joan Navarro**, Xavi Canaleta, Xavi Salada. *LSscheduling: Herramienta interdisciplinar de soporte docente para la asignatura de sistemas operativos.* XVIII Jornadas de Enseñanza Universitaria de la Informática (JENUI 2012). July 10-13, 2012, Ciudad Real, Spain. Conference proceedings.

- Jordi Albo-Canals, **Joan Navarro**, David Serra-Puig, Xavier Vilasís-Cardona. *A robot swarm as a cellular multicore processor.* IEEE International Symposium on Circuits and Systems (ISCAS 2012). May 20-23, 2012. Seoul, Korea. Conference Proceedings IEEE.

- José Enrique Armendáriz-Íñigo, Itziar Arrieta-Salinas, **Joan Navarro**, August Climent. *Transactional support on the cloud. Taking advantage of classical approaches.* The 1st DEXA Workshop on IT Service Management and its Support (ITSMS 2011). August 29, 2011, Toulouse, France. Conference Proceedings. IEEE-CS Press.

- **Joan Navarro**, José Enrique Armendáriz-Íñigo, August Climent. *Dynamic distributed storage architecture on Smart Grids.* The 6th International Conference on Software and Data Technologies (ICSOFT 2011). July 18-21, 2011, Seville, Spain. Conference Proceedings. INSTICC Press.

- **Joan Navarro**, Xavi Canaleta, Andreu Sancho-Asensio. *Sistemas operativos avanzados: De la clase magistral al entorno colaborativo.* XVII Jornadas de Enseñanza Universitaria de la Informática (JENUI 2011). July 5-8, 2011, Seville, Spain. Conference Proceedings.

- **Joan Navarro**, Ainhoa Azqueta, Pablo Murta, José Enrique Armendáriz-Íñigo. *Cloud computing keeps financial metrics computation simple.* The 6th International Conference on Software and Data Technologies (ICSOFT 2011). July 18-21, 2011, Seville, Spain. Conference Proceedings. INSTICC Press.

- **Joan Navarro**, José Enrique Armendáriz-Íñigo, August Climent. *Cloud-based replication protocol on power Smart Grids.* Workshop on Software Services: Cloud Computing and Applications based on Software Services (2nd WOSS). June 6-9, 2011, Timisoara, Romania. Conference Proceedings.

- Xavi Canaleta, **Joan Navarro**. *Herramientas de soporte al aprendizaje de sistemas de ficheros.* XVI Jornadas de Enseñanza Universitaria de la Informática (JENUI 2010). July 7-9, 2010, Santiago de Compostela, Spain. Conference Proceedings, p. 413-419, ISBN: 846933741-6.

- August Climent, **Joan Navarro**, Miquel Bertran, Francesc Babot. *Optimistic concurrency control with partial replication design.* International Conference on Applied Computing (IADIS 2009). November 19-21, 2009, Rome, Italy. Conference Proceedings.

Non-peer reviewed conference papers

- Mariela Louis-Rodríguez, Andreu Sancho-Asensio, **Joan Navarro**, Itziar Arrieta-Salinas, Ainhoa Azqueta-Alzúaz, José Enrique Armendáriz-Íñigo. *A prospective view on designing partitioning schemes for replicated databases.* XXI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2013). June 19-21, 2013, San Sebastián, Spain. Conference Proceedings.

- Itziar Arrieta-Salinas, **Joan Navarro**, José Enrique Armendáriz-Íñigo. *Providing transactional support on the cloud: Hybrid approaches.* XX Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2012). June 13-15, 2012, Pamplona, Spain. Conference Proceedings.

- **Joan Navarro**, José Enrique Armendáriz-Íñigo, August Climent. *Highly scalable replication protocol on power Smart Grids.* XIX Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2011). June 8-10, 2011, Segovia, Spain. Conference Proceedings.

- **Joan Navarro**, August Climent, Miquel Bertran, Francesc Babot. *Optimistic concurrency control with abort prediction.* XVII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2009). June 10-12, 2009, Sagunto, Spain. Conference Proceedings.

Technical reports

- **Joan Navarro**, August Climent, Miquel Bertran, Francesc Babot. *Optimistic concurrency control with abort prediction and partial replication.* Technical report DI-09003-01-GRSD. La Salle, Universitat Ramon Llul. Barcelona, 2010.

In the end, though, maybe we must all give up trying to pay back the people in this world who sustain our lives. In the end, maybe it's wiser to surrender before the miraculous scope of human generosity and to just keep saying thank you, forever and sincerely, for as long as we have voices.

— Elizabeth Gilbert, 2007.

ACKNOWLEDGMENTS

I consider myself very lucky to have met such a wide variety of people—from different cultures and regions around the world—that have made me grow personally and professionally in the course of this long journey. Along this trip, there are three outstanding people that deserve my deepest gratitude and appreciation. First, Núria Macià, who bravely encouraged me to *escape forward from the formal classicism* and had the stubbornness to teach me everything I know about Research. There are no words to acknowledge all what Núria has done and meant for me, I will be always indebted to her. Second, Itziar Arrieta, who despite her cultural heritage never had a *no!* for me. She demonstrated an unwavering devotion on coding, writing, reviewing, and discussing *alternative approaches*, which showed me that I have still a lot to learn and makes me feel fortunate to be her colleague and friend. Third, Enrique Armendáriz the *ancient man inside a teenager body*, who honestly trusted in me from the very beginning, unlocked several doors, and adopted me as his grandson. I feel immensely honored to have received his valuable lessons about databases and, most important, about life philosophy.

My thankfulness also must be extended to August Climent, who was undoubtedly the first to awaken my interest towards distributed databases. Indeed, he has had an important role during all these years by introducing me to the appropriate people, giving me the freedom to make my own way, and providing me with the fundamentals to successfully conduct this research.

I would also like to thank my teachers and coworkers at La Salle, Universitat Ramon Llull for shaping me as an Engineer and making my days at the engineering department joyful. Together with them, I would like to appreciate all my students and teaching assistants, without their help, this would not have been possible. Particularly, Carmen Martínez deserves a special recognition for her support during all this period, for flushing my mind every time it got congested, and for showing me the beauty of other things in life besides academia.

Last, but not least, I am very grateful to Fernando de la Torre for allowing me to do part of this research at the Robotics Institute in Carnegie Mellon University. Definitely, this work has greatly benefited from daily interactions with colleagues at the Human Sensing Lab.

FUNDING

This research has been financially supported by the Spanish Ministry of Economy and Competitiveness under grant TIN2012-38416-C03-02, European Union European Atomic Energy Community Seventh Framework Programme (FP7/2007-2013 FP7/2007-2011) under grant agreement 247938, Programa Nacional de Tecnologías de la Información y las Comunicaciones, Ministerio de Educación y Ciencia with reference TIC2006-14738-C02-02, and Enginyeria i Arquitectura La Salle – Universitat Ramon Llull.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

1CMV	1-Copy Multi-Version
1CSI	1-Copy Snapshot Isolation
1SR	1-Copy Serializability
2PC	2-Phase Commit
2PL	2-Phase Locking
AAA	Authentication, Authorization, and Accounting
ACID	Atomicity, Consistency, Isolation, and Durability
ANM	Active Network Management
ANSI	American National Standards Institute
API	Application Programming Interface
BASE	Basically Available, Soft state, and Eventually consistent
BB-PLC	Broad Band Power Line Communications
BOCC	Basic Optimistic Concurrency Control
BRP	Basic Replication Protocol
CaaS	Consistency as a Service
DBMS	Database Management System
DFD	Data Flow Diagram
DMA	Domain Management Agent
DNS	Domain Name Service
DRY	Don't Repeat Yourself
ETL	Extract, Transform, and Load
FIFO	First In First Out
GCS	Group Communication System
GFS	Google File System

HDFS	Hadoop Distributed File System
ICT	Information and Communications Technology
I-Dev	INTEGRIS Device
IDE	Integrated Development Environment
IED	Intelligent Electronic Device
IEEE	Institute of Electrical and Electronics Engineers
INTEGRIS	INTelligent Electrical GRId Sensor Communications
JDBC	Java Database Connector
LCS	Learning Classifier System
NDCC	Non-Distributed Concurrency Control
NoSQL	Not only SQL
O2PL	Optimistic Two Phase Locking
ODBC	Open Database Connector
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
OSI	Open Systems Interconnection
PAA	Perception-Action Agent
PADD/RALE	Parallelism and Abstraction Distributed Design – the RAmon Llull Environment
PLP	PhysioLogical Partitioning
QoS	Quality of Service
RAWA	Read All Write All
RDBMS	Relational Database Management System
ROWA	Read One Write All
RS	Read Set
S2PL	Strict Two Phase Locking
SABI	Sistema de Análisis de Balances Ibéricos
SQL	Structured Query Language
STP	Spanning Tree Protocol
TCP	Transport Control Protocol

TLS	Transport Layer Security
tps	transactions per second
UCS	sUpervised Classifier System
UML	Unified Modeling Language
WAN	Wide Area Network
WORM	Write Once and Read Many
WS	Write Set
XCS	eXtended Classifier System
XML	eXtensible Markup Language
YCSB	Yahoo! Cloud Serving Benchmark

1 INTRODUCTION

Summary. Data storage techniques have evolved along the history to satisfy different demands and achieve different levels of scalability. However, interesting features, such as transactional support in highly scalable architectures, have been neglected on these advances, which prevents several applications to benefit from them. In fact, current implementations attempt to smartly avoid issues found in the past in order to meet the requirements faced in the present and (near) future. This chapter (1) reviews how data storage has been addressed so far—ranging from centralized raw storage computers and transactional replicated databases to cloud-based data repositories—and (2) introduces the goals of this thesis.

"Ask not what your country can do for you, ask what you can do for your country"
— John Fitzgerald Kennedy, 1961.

1.1 INTRODUCTION

In the early seventies, mathematicians and engineers aimed at building powerful computers—in terms of computing capabilities—by increasing the number of operations per second a microprocessor could solve. However, a few years later they realized that computers could also be used as digital data warehouses, which could provide appealing features that traditional paper-based files were unable to offer [Inmon, 1981], such as multi-indexing, reduced information-to-space ratio, or advanced privacy policies. Unfortunately, the amount of digitalized data—from paper documents to multimedia files—grew faster than the storage technology developed. Standalone computers with few megabytes of hard disk capacity were not able to meet the storage requirements of such demands. It was not until the growth of the Internet and the evolution of computer networks that it became possible to build computer clusters (also referred to as computer farms) and overcome the reduced storage facilities of single machines [Brownbridge et al., 1982].

Clusters were targeted at hiding the internal architecture and configuration of each single computer within the cluster (i.e., physical location, system capabilities, current load, available resources, etc.) and offered a general purpose data repository with outstanding storage specifications. However, computer clusters—strongly dependent on an unreliable network backbone—were fault prone and poorly scalable. This raised a new challenge concerning data indexing, computing and processing. In addition, availability and reliability user requirements towards data repositories became mandatory since several third party critical applications depended on these data. These concerns drove the research community to explore new strategies for dealing with data storage.

1.1.1 *Replication in distributed databases*

Two major alternatives were proposed to face data availability and reliability: (1) distributed file systems—aimed at storing raw data—(out of the scope of this dissertation) and (2) distributed databases (also referred to as cluster databases)—aimed at storing indexed data and executing high level queries.

Replication was the first approach to improve the throughput provided by a single site [Bernstein et al., 1987]. It consists in placing the same data item into several machines within the cluster [Pedone et al., 2000]. This strategy allowed distributed databases to be fault tolerant since data stored on a single site was not critical any more (i.e., it was already saved in neighboring sites). Furthermore, a technique to allow simultaneous access over the replicated and fault tolerant database (i.e., improving the aforesaid data availability) consisted in the usage of transactions, which were defined as indivisible units that grouped a set of operations and observed the Atomicity, Consistency, Isolation, and Durability (ACID) properties [Härder and Reuter, 1983; Bernstein et al., 1987; Brewer, 2012]. Transactions provided an effective mechanism to (1) encapsulate critical operations, (2) ensure reliable concurrent access to data, and (3) access distributed databases which hold complex data structures, schemas, or data models. Nowadays, they can be easily built and driven through a Structured Query Language (SQL) which links the application layer with the data storage layer.

In fact, replication next to transactions permitted that several machines could serve different read requests (also referred to as queries) at a time and thus, (ideally) multiplied the global system throughput up to the number of sites in the farm. On the contrary, write operations (also referred to as updates) became extremely time consuming, since data needed to be written on every site [Gray et al., 1996]. This issue led to hybrid solutions which replicated data up to a certain number of sites (also referred to as partial replication). These attempts improved the write operations cost and reduced the fault tolerance and data availability properties [Serrano et al., 2007; Bernabé-Gisbert et al., 2008]. However, there is still an open discussion on how to optimize the trade-off between replication, data reliability, and availability [Kemme and Alonso, 2000a; van Steen and Pierre, 2010; Krikellas et al., 2010; Brewer, 2012].

To reduce the cost of propagating updates to all replicas, two different strategies have been proposed concerning when replicas are updated [Wiesmann and Schiper, 2005]: (1) eager replication and (2) lazy replication.

- **Eager replication** consists in blocking incoming operations until all replicas acknowledge to have received the update operation—which intrinsically ensures data consistency properties but limits the throughput [Krikellas et al., 2010].
- **Lazy replication** consists in allowing incoming operations to progress without any guarantees that previous updates have been successfully applied to all replicas—which threatens data consistency properties [Serrano et al., 2007].

These two strategies can be combined with two other techniques concerning where replicas are updated [Plattner, 2006]: (1) primary copy and (2) update-everywhere.

- **Primary copy** consists in forcing all clients to execute update operations on the same replica (also referred to as primary copy) and let it propagate the changes to other replicas—which eases replica convergence but exhibits the single point of failure phenomenon [Alsberg and Day, 1976; Pedone, 1999].

- **Update-everywhere** consists in allowing clients to execute update operations on any copy—which adds an extra burden to the system due to the distributed synchronization process carried between all replicas to synchronize concurrent updates executed at different replicas (also referred to as conflicts) [Plattner et al., 2008].

Hence, the more replicated data are, the more research challenges arise [Gray et al., 1996]. Once availability seemed to be fairly solved by replication, data consistency started being threatened. In order to keep all data items consistent within the database (i.e., all sites contain the same value of a given data item) a synchronization protocol (also referred to as concurrency control) between all replicas was required [Kung and Robinson, 1979; Thomasian, 1998a; Al-Jumah et al., 2000].

1.1.2 *Concurrency control strategies*

A complete transaction is aborted as soon as the concurrency control manager detects that an operation inside the transaction violates any datum consistency property (i.e., there is another transaction accessing the same item). This action allows other transactions to progress subsequently. This is carried by a synchronization protocol—in charge of maintaining the aforementioned ACID properties in all replicas—that is typically message-based and consists in exchanging the status of each datum to check the correctness of the operations applied to it. This verification adds a considerable overhead to the system. Actually, these expensive protocols degrade the global system throughput because data operations have to wait until all synchronization messages ensure that there is no other operation accessing any data item contained inside each transaction [Gray et al., 1996]. Although there are specific scenarios where the ACID restrictions can be relaxed (e.g., mail lists, Domain Name Service (DNS) servers, web sites) and hence the synchronization protocol overhead can be drastically relaxed, data replication next to its consistency properties and concurrency control are still being a hot research topic.

Optimistic concurrency control and pessimistic concurrency control are the two main strategies [Bernstein et al., 1987; Thomasian, 1998b] to slightly minimize the effects of this overhead.

- **Optimistic concurrency control** allows operations contained inside a transaction to be executed with no restrictions and checks for consistency violations at its end—useful when access conflicts are improbable.
- **Pessimistic concurrency control** checks that no conflictive situations will exist prior executing every operation—which is more suitable for conflict intensive landscapes.

Despite the flexibility of both strategies, there is still a big limitation concerning the scalability of the aforesaid distributed databases. As data volumes and number of sites increase, operations contained inside the same transaction demand accessing data from many distant sites, which sharply drowns the whole communication network and thus the global database throughput.

An alternative approach to alleviate the stringent scalability of such databases consists in relaxing some of the ACID properties and transferring their management to high-level layers—which forces programmers to be aware of data storage specifications but gives them enough freedom to build customizable applications. In fact, it is possible to define different data isolation levels to determine how and when changes made by an operation become visible to the other concurrent ones, and thus reduce the overhead of the concurrency control protocol and enhance the system performance [Elnikety et al., 2005; Serrano et al., 2007; Berenson et al., 2007]. It is also possible to define a new form of consistency, coined as weak consistency [Vogels, 2009], that allow the execution of conflicting operations in some particular situations—although this increases the distributed system throughput.

Note that all of these works that are suitable for specific scenarios have an upper bound. It has been proved that it is not possible to have full data availability, strong data consistency, and perfect network partition tolerance at the same time in any networked shared-data system. Actually, there is a trade-off (referred to as Brewer's CAP theorem [Brewer, 2000, 2012]) between all of them [Brewer, 2000, 2012]—which was formally confirmed by Gilbert and Lynch [2002]. Designers have assumed this limitation and, reasonably powerful applications with fairly acceptable performance have been developed according to such constraints (e.g., stock exchanges, tickets booking).

However, the ever-growing data volumes and the rising storage demands—no longer affordable by these transactional systems—have led the research community to go beyond Brewer's constraints and find out the way to overcome such restrictions. It is known that transactions may limit the scalability of a distributed database [Ryu and Thomasian, 1990]. Thus, it is recommended to make them as short as possible to lock as less data items as possible (i.e., reduce I/O disk requests, minimize network stalls, avoid interactive transactions) [Johnson et al., 2012] and allow concurrent transactions progression. Practitioners have brought this statement to the limit and defined a new concept of highly scalable database with single operation transactions, known as key-value stores [DeCandia et al., 2007; Das et al., 2010a]. This new storage paradigm (also referred to as Not only SQL (NoSQL)) renounces from the richness of relational algebra and joined tables associated to transactions in exchange for high scalability features [Cattell, 2010; Stonebraker, 2010].

1.1.3 *Cloud-based storage*

Storage based on big key-value tables has been named as cloud-based storage. The cloud paradigm involves further concepts such as (virtually) infinite scalability and services on demand over a multi-tenant architecture [Kraska et al., 2009b]. However, the underlying idea is to:

1. Relax data consistency by relying on weak consistency models [Vogels, 2009; Bernstein and Das, 2013], which do not guarantee that read operations performed after an update will return the last updated value. For instance, eventual consistency—which is the most common particular case of weak consistency—states that if no new updates are applied to a data item, eventually all accesses will return the last updated value.

2. Constrain data durability features to minimize the input/output disk access (also referred to as soft state systems).

3. Reduce the overhead associated to the concurrency control manager [Ghemawat et al., 2003; Chang et al., 2008; White, 2011; Baker et al., 2011].

In this approach, data forfeit their ACID properties but satisfy what has been coined as Basically Available, Soft state, and Eventually consistent (BASE) properties [Fox et al., 1997; Brewer, 2012]. Although this new property definition has widely been accepted, there are still several applications (e.g., bank transferences, on-line reservations) that do need ACID properties due to their intrinsic transactional nature, and thus cannot take benefit from the advantages of these NoSQL approaches. These applications have to carry on with the classic database storage strategies which prevent themselves to scale up and provide modern functionalities [Bouhafs et al., 2012]. In fact, moving those applications (i.e., providing them with transactional support) to a cloud-based repository is still challenging.

This thesis is concerned about advancing in the research on distributed storage repositories—from the classic to the cloud-based ones—to gain a better understanding of their behavior and to improve them to deal with current problems in industry. Further, we identify two important challenges in providing transactional support on the cloud (1) scalability limitations of transactional databases and (2) transactional limitations of cloud-based repositories, which are later articulated in the objectives of this work. Finally, we provide the road map of the entire document.

1.2 MODERN CHALLENGES IN DISTRIBUTED DATABASES

Research on distributed databases has lately resulted in the design of concurrency control protocols and replication techniques that can light the overhead associated to the concurrency control manager and the replication process. Nevertheless, the database community has recently had to address the crucial challenge that appears when trying to manage huge amounts of data (also referred to as Big Data) typically found in current real-world applications (e.g., Google, Facebook, Amazon, Twitter). Among the different research lines, the following two challenges have received especial attention:

1. Scalability in transactional distributed databases.

2. Transactional support on cloud-based storage repositories.

A more detailed discussion on why these two items are critical aspects in data management is provided as follows.

Scalability in transactional distributed databases. Classic distributed databases store and retrieve structured data in an effective way. Data are replicated over an arbitrary set of servers in order to provide high availability and fault tolerance. If strong consistency [Bernstein et al., 1987; Vogels, 2009; Krikellas et al., 2010] is demanded—transactional systems usually do—the more servers are added to the system, the more replicas have to be kept synchronized. Hence, the system spends considerable efforts in the replication process instead of the transactions' execution [Gray et al., 1996]—which penalizes its throughput. Furthermore, transactions have the freedom to access any datum stored in whatever server the cluster database is deployed, which prevents the system from being partitioned (also referred to as sharding [Indelicato, 2008]). Hence, in these kinds of systems where neither support for network partitions nor consistency can be relaxed, data availability has to be sacrificed [Brewer, 2000; Gilbert and Lynch, 2002; Brewer, 2012]. Paradoxically, when adding more servers to improve data availability, it becomes less available due to the replication process and concurrency control protocol [Gray et al., 1996].

While this problem has been widely studied in the context of traditional distributed databases, little research on moving this concern to cloud storage repositories has been conducted. So far, several proposals have only aimed at smartly evading such constraints [Brewer, 2012] under certain circumstances in the context of cluster databases; for example, optimistic concurrency control [Bernstein et al., 1987], lazy replication [Wiesmann and Schiper, 2005], partial replication [Serrano et al., 2007; Bernabé-Gisbert et al., 2008; Armendáriz-Iñigo et al., 2008; Peluso et al., 2012], smart data placement [Curino et al., 2010], smart partitioning [Pandis et al., 2011b], transaction decomposition [Pandis et al., 2011a], write ahead loggings [Johnson et al., 2012], or eventual consistency [Burckhardt et al., 2012; Bernstein and Das, 2013]. However, few successful attempts have been proposed to develop a scalable transactional distributed database solution general enough to be adaptable at each situation without losing the transactional semantics [Johnson et al., 2012]. This lack of scalability prevents industrial applications that demand both On-Line Analytical Processing (OLAP) and On-Line Transaction Processing (OLTP) facilities (e.g., Web 2.0) to be run in cluster databases [Cao et al., 2011].

Transactional support on cloud-based storage repositories. Cloud-based storage repositories are currently developed by the NoSQL community [Cattell, 2010]. They are designed to satisfy high availability demands and run under elastic conditions—resources addition and removal (e.g., partition, servers, replicas) according to dynamic load requests. To achieve such commitment, the richness of relational algebra and transactional capabilities have been moved away [Ghemawat et al., 2003; DeCandia et al., 2007; Baker et al., 2011] in several approaches. However, there are still many situations cannot resign from their transactional nature and strong consistency requirements but claim for the advantages featured by the cloud paradigm. Actually, providing consistency, availability, and network partition tolerance is not totally against the Brewer's theorem [Brewer, 2012]. These properties can be individually relaxed and thus it is possible to design a system able to provide relaxed (also referred to as weak) consistency, acceptable availability, and reasonable network partition tolerance at a time. In fact, first steps exploiting this idea have been conducted by either reducing the transactions' scope or relaxing the data consistency properties. For instance, Sinfonia [Aguilera et al., 2009] implements a highly scalable system that uses a special form of transactions, coined as mini transactions. Mini transactions reduce the operations' coupling by (1) considering distributed shared memory as a service and (2) resizing transactions—thus reducing the overhead associated to the replication process. Indeed, long transactions typically generated by SQL sentences cannot directly be run over this system. Kraska et al. [2009b] suggested that it is also possible to ration data, instead of transactions, into several consistency categories and use temporal statistics to properly balance transactions in a cloud scheme to optimize the total operational costs. Nevertheless, this approach requires big efforts in terms of configuration with regard to the statistical methods and consistency levels. ElasTraS [Das et al., 2010b] is an experimental system that uses the cloud-based storage system Apache Hadoop Distributed File System (HDFS) [White, 2011] as an underlying layer to implement a distributed database for a partitioned multi-tenant architecture. This system inherits the HDFS high scalability and implements a load balancer and a transaction manager on top of it. Despite the high flexibility offered to the end-users, practitioners do not have a full control of the storage layer and, hence, data durability might be compromised. Recently, Burckhardt et al. [2012] have explored eventually consistent transactions—as cloud systems do with raw data—and consequently improved the scalability of relational databases. However, despite the latest attempts [Curino et al., 2011b; Vo et al., 2010], there does not exist any complete solution able to provide pure transactional support on a cloud infrastructure with full storage layer control as Relational Database Management Systems (RDBMSs) do.

Therefore, the scope of this thesis is to address these two challenges in the context of realistic applications (e.g., the Smart Grid [Bouhafs et al., 2012]). We consider transactional distributed databases as starting point since they own the essential functionalities we want to export to the cloud. Besides, we incorporate the most influential and state-of-the art representative cloud-based storage repositories to set out the basics of this thesis. We design and implement a novel storage architecture based on the cloud paradigm. The following section articulates in more detail the objectives of this thesis.

1.3 THESIS OBJECTIVES

Modern challenges regarding scalability in transactional distributed databases and transactional support on the cloud lead to the definition of the following four objectives:

1. Revise concurrency control protocols and replication techniques to compare their performance.
2. Analyze cloud-based storage repositories.
3. Apply the cloud storage paradigm to solve current scalability issues in industry.
4. Design and implement a storage infrastructure able to provide transactional support on the cloud.

In the following, each objective is elaborated.

Revise concurrency control protocols and replication techniques to compare their performance. Over the last few decades, concurrency control and replication protocols have extensively been investigated and brought important advances in the field of distributed databases. Most of such advances have not been used, however, in commercial products [Kemme and Alonso, 1998]. They have neither been formally compared nor assessed under the same conditions due to their associated complexity [Kemme and Alonso, 2000b]. In addition, partial replication layouts still present challenging issues to the study of these protocols [Serrano et al., 2007; Bernabé-Gisbert et al., 2008]. Therefore, this thesis provides a general-purpose simulation framework to run any concurrency control and replication protocol and analyze its behavior under a unified testing environment. We also explore partial replication concurrency control algorithms with this framework as a very first approach to cloud-based multi-tenant systems.

Analyze cloud-based storage repositories. We review the most relevant cloud-based storage repositories in the literature to find out the main, most appealing components of these approaches [Agrawal et al., 2011]. Specifically, we study the intrinsic properties of NoSQL systems to identify and scale critical factors for their success. The analysis goes from semi structured tables [Harter et al., 2014] to raw data storage [White, 2011], including key-value stores [DeCandia et al., 2007], and allows us to draw the domain of applicability of each system. Moreover, we aim at extracting lessons from this analysis that help to set out the basics of providing transactional support on the cloud.

Apply the cloud storage paradigm to solve current scalability issues in industry. After studying the most significant cloud-based storage repositories, we aim at testing their performance when facing real-world problems. First, we assess their behavior using existing approaches. And then, we use the results of the analysis to design and implement a custom key-value store based on the cloud storage paradigm to solve the critical storage problem entailed by Smart Grids [Bouhafs et al., 2012]—also common in other application domains.

Design and implement a storage infrastructure able to provide transactional support on the cloud. We address the second aforementioned challenge and take an alternative approach to mix the ideas of transactional databases and cloud storage fields. That is, we create a hybrid system that combines the best features from classic distributed databases—as transactions and ACID properties—, replication techniques—as an effective technique to provide different levels of consistency—, and highly scalable storage repositories—as a powerful landscape to deploy elastic services.

The overall structure of the document is provided in the following section.

1.4 ROAD MAP

The dissertation is arranged in eight chapters whose content is introduced in what follows.

Chapter 2. **Transactions processing in replicated databases** introduces the fundamentals of transaction processing, replication techniques, group communication systems, and classic distributed databases layout. This leads to the proposal of an abstract model that includes all these topics and will be used to build a simulator of a transactional database.

Chapter 3. **Performance analysis of concurrency control and replication protocols** proposes a simulator for transactional distributed databases and provides a detailed description of the most relevant concurrency control and replication techniques. In addition, it compares their performance over the simulation environment proposed in the previous chapter and justifies their scalability limitations.

Chapter 4. **Cloud storage: Boosting data storage scalability** explores the state-of-the-art concerning cloud storage and provides a set of arguments that attest the high scalability of such systems. Furthermore, it describes the out of the box experience of one of the most representative cloud storage systems—Apache HDFS—when facing real-world problems. Additionally, it unveils some limitations for its application to other domains.

Chapter 5. **A custom approach to large-scale key-value stores** proposes a first attempt of transactional support on the cloud. It presents the design and implementation of a cloud inspired key-value store infrastructure in the context of the Smart Grid, which also gathers some ideas of transactional replicated databases.

Chapter 6. **A cloud-based infrastructure for power Smart Grids** proposes a distributed Information and Communications Technology (ICT) infrastructure—based on the key-value storage infrastructure depicted in the previous chapter—especially designed to face the real-world Smart Grid singularities. This ICT approach includes a reliable network communications system, a security and trusting management module, a distributed storage layer, and a cognitive system.

Chapter 7. **Providing transactional support on the cloud** presents Epidemia, a novel hybrid architecture that combines transactional databases and cloud storage. Inspired by the key-value store described in the previous chapters, it includes several new modules which allow it to effectively deal with transactions and inherit the high scalability of the cloud underlying architecture. An analytical model, a formal correctness proof, and a prototype implementation of Epidemia are presented to show its behavior under different workloads and demonstrate that its capabilities on alleviating the scalability limitations of traditional replicated databases.

Chapter 8. **Summary, conclusions, and further work** concludes the dissertation by summarizing the contributions of this thesis, providing key conclusions on the achieved results, and pointing future research directions.

2 TRANSACTIONS PROCESSING IN REPLICATED DATABASES

Summary. Transactions provide programmers and application developers with a powerful mechanism to deal with concurrency issues. They guarantee data isolation and keep data consistency in the context of a distributed database as long as they encapsulate an atomic set of read/write operations. Notwithstanding, they also limit the scalability of cluster databases since they prevent data from being fully decoupled. This chapter analyzes transactions processing in replicated databases and proposes an abstract model to later simulate their behavior and find out additional improvements.

"Simplicity is the ultimate sophistication"
— William Gaddis, 1955.

2.1 INTRODUCTION

Distributed databases are a particular case of distributed systems [Tanenbaum and Steen, 2006] and are composed of network communications and (distributed) data processing. They present a suitable context to store, retrieve, and process structured data in order to overcome the limitations derived from the usage of a single computer [Bernstein et al., 1987]. Özsu and Valduriez [1999] distinguish three types of distributed databases depending on what is being distributed: (1) processing logic or/and processing elements distribution (i.e., applications are distributed in different sites) [Birman, 2012], (2) function distribution (i.e., hardware or software functions of a given site are assigned to other sites) [Stonebraker et al., 2010], and (3) data distribution (i.e., data is replicated across several sites) [Fekete and Ramamritham, 2010].

This dissertation focuses on data distribution since it is not directly related with any specific application. This kind of databases—also named replicated databases—and their architecture have lots of similarities with shared-nothing architectures proposed in the distributed systems field [Özsu and Valduriez, 1999; Tanenbaum and Steen, 2006; Das, 2011; Birman, 2012]. The main difference between both is that replicated databases do not require a symmetric resource assignment in all sites. Therefore, a replicated database[1] can be seen as a set of servers that host the same data and execute a set of primitives (i.e., transactions) over them.

In this chapter, we first introduce basic concepts in distributed databases related to transaction processing, then briefly overview existing data replication strategies and group communication techniques, and later describe how the system correctness can

1 Replicated database, cluster-based, and distributed database will be interchangeably used along the document.

be theoretically assured in the context of a database. Then, we decompose the layout of traditional Database Management Systems (DBMSs), which will set the basis of our distributed database abstract model that introduces the working environment proposed in the following chapter.

2.2 ACCESS TO DATA IN DISTRIBUTED DATABASES

There is a set of principles and good practices that designers have to take into account when designing a distributed database or an RDBMS [Inmon, 1981; Bernstein et al., 1987]. For instance, Das [2011] suggests:

1. Separating system state from application state.
2. Decoupling data storage from ownership.
3. Limiting common operations to a single node.
4. Limiting distributed synchronization.

Most of these recommendations are basically aimed to isolate the data layer from the application layer. This allows each side to grow independently and, thus, foster their scalability. In addition, these recommendations help to implement data concurrency and permit several applications, or even many different instances from the same application, safely access the same data (also referred to as concurrency).

In order to link the data plane with the application domain, these guidances stand for a vehicular instrument—standard and versatile enough to be used by any application and allow effective access towards data—to link both layers [Alonso et al., 1996]. From the application side, this is generally named Open Database Connector (ODBC)—or Java Database Connector (JDBC) in the specific case of Java-based applications—and holds complex high-level sentences to access data (e.g., `SELECT * FROM tblMyTable WHERE timestamp > 69`). On the contrary, from the data side those sentences issued through the high-level connectors are transformed in what it is known as transactions [Bernstein et al., 1987], which contain a finite set of simple read/write operations to be performed over an object. Without loss of generality, the scope of this thesis is concerned in the data side and, thus, in transactions, which are elaborated in what follows.

2.2.1 *Transactions*

Transactions—introduced with the earliest database system models [Bernstein et al., 1987]—are a standard interface to store, update, and query data through a high level query language running on the application layer [Traiger et al., 1979; Gray, 1980]. Nevertheless, their primary goal is to make the concurrency control and fault recovery easier [Bernstein and Goodman, 1981; Schiper and Raynal, 1996] by encapsulating a finite set of operations and satisfying all ACID properties [Härder and Reuter, 1983; Bernstein et al., 1987; Adya, 1999]:

1. Atomicity. It claims that if one part of the transaction (considered as an indivisible set of operations) fails then the entire transaction fails, and thus the database state is left unchanged (also referred to as all-or-nothing property).

2. Consistency. It claims that any transaction must bring the database from one valid state to another—ideally, all replicas of the same item contain the same value at any time.

3. Isolation. It claims that operations executed in a transaction must be hidden from other transactions running concurrently. Otherwise, a transaction could not be reset to its beginning in case of failure.

4. Durability. It claims that the effect of the performed operations contained inside the transaction is permanent.

In order to better understand the criticalness of these attributes, they can be generalized into three properties if consistency and isolation are put together into a new property called ordering. The ordering property asks that all transactions have to be executed in some sequential order (also referred to as correctness) [Guerraoui and Schiper, 1994]. Therefore, to provide applications with the expected outcomes (e.g., variable x equals to $x+1$ after it is incremented) critical operations have to be (1) encapsulated inside a transaction and executed atomically (read value from variable x and update it accordingly with $x+1$), (2) executed without interfering in other applications execution, and (3) permanently stored in memory to allow subsequent transactions to work with previous results.

Transaction manager is the database entity in charge of executing transactions and guaranteeing that operations are executed in a safely and reliable way [Bernstein et al., 1987]. In order to synchronize the tasks of the transaction manager, there is a standard set of commands that each site of the database should use to atomically execute the operations. On the one hand, there are the retrieve and store operations [Bernstein et al., 1987]:

1. Retrieve: `read(object)` (also referred to as "$\texttt{r}(X_i)$", for $X_i \in$ `database`) returns the stored item pointed by `object`.

2. Store: `write(object,val)` (also referred to as "$\texttt{w}(X_i,V_i)$", for $X_i \in$ `database and` $V_i \in X_i$ `domain`) updates the value of item `object` (X_i) to the value of V_i.

Note that depending on the granularity level of the system, X_i can either refer to a datum, a row, a table, or even a database.

On the other hand, there is a set of synchronization operations [Bernstein et al., 1987]:

1. Begin: `begin()` (also referred to as "`b`") indicates the beginning of a transaction.

2. Commit: `commit()` (also referred to as "`c`") indicates a successful termination of a transaction. At this point, all operations contained inside the transaction are executed and their results are permanent.

3. Abort: `abort()` (also referred to as "`a`") indicates a failed termination. None of the operations contained inside the transaction are executed.

Therefore, each SQL sentence issued from the application layer is translated into one or more streams of read/write operations enclosed within a `begin()` and either a `commit()` or an `abort()` (see Example 1).

Example 1. *This example illustrates two independent transactions* T_0 *and* T_1 *that both read from the object* X*. The first one updates the object with the value* 33 *and the second one updates the object* Y *with the value* 69*. Finally, the first transaction commits and the second one aborts.*

$$T_0 = \{b_0, r_0(X), w_0(X,33), c_0\},$$
$$T_1 = \{b_1, r_1(X), w_1(Y,69), a_1\}.$$

The operation subscript let the system know which transaction is the owner of each operation. This has special interest when executing many transactions concurrently. The following section discusses further consequences derived from executing several transactions simultaneously.

2.2.2 *Concurrency*

Concurrency protocols address the underlying effects associated to the execution of two or more transactions [Lamport, 1978a]. Critical issues arise when operations from different transactions attempt to access the same object [Bernstein and Goodman, 1981] (see Example 2).

Example 2. *This example illustrates two transactions* T_0 *and* T_1 *that both read from the object* X *and update it accordingly.*

$$T_0 = \{b_0, r_0(X), w_0(X, X+33), c_0\},$$
$$T_1 = \{b_1, r_1(X), w_1(X, X+69), c_1\}.$$

In the previous example, if we assume that $X = X_0$ before their execution and T_0 and T_1 are executed concurrently—without concurrency control—, X may have different possible values at the end of the execution (also referred to as history H [Bernstein et al., 1987]) depending on the operations' interleaving:

$$H_1 = \{b_0, b_1, r_0(X), w_0(X, X+33), r_1(X), w_1(X, X+69), c_1, c_0\},$$
$$H_2 = \{b_0, b_1, r_0(X), r_1(X), w_0(X, X+33), c_0, w_1(X, X+69), c_1\},$$
$$H_3 = \{b_0, b_1, r_0(X), r_1(X), w_1(X, X+69), c_1, w_0(X, X+33), c_0\},$$

which may lead to $X = X_0 + 33 + 69$ for H_1, $X = X_0 + 69$ for H_2, and $X = X_0 + 33$ for H_3 respectively. In general, this kind of interleaving is difficult to control from the application side since it is loosely coupled with low layer storage. Thus, the concurrency control manager—which belongs to the transaction manager of the DBMS—is in charge of avoiding malicious interferences and incorrect application outcomes [Bernstein et al., 1987] according to the database layout and application requirements.

For instance, a centralized database (i.e., a database with a single server [Bernstein et al., 1987]) may be able to ensure safe data accesses by only satisfying atomicity and durability properties without further concurrency control manager assistance if (1) transactions are properly built and (2) semaphores, critical regions, or other mutual exclusion mechanisms are used at the application layer. Indeed, despite the fact that several applications may compete for the same storage resources (also referred to as multi-tenancy [Das, 2011]), there is a single local entity in charge of deterministically serializing these requests in form of transactions and, thus, avoiding unexpected and misleading. This means that in this case all operations are always executed sequentially.

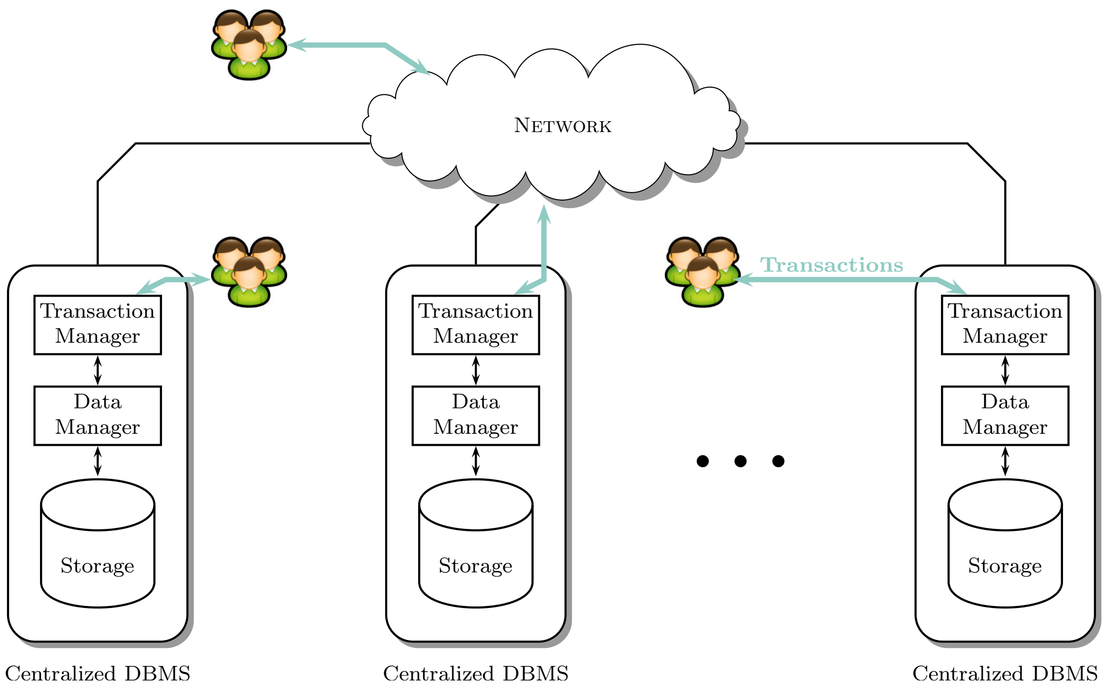


Figure 2.1: Conceptual model of a distributed database. Clients issue transactions (green arrows) against any transaction manager of a single node in the database.

On the contrary, some challenging issues appear when generalizing the centralized model to a shared-nothing database architecture [Das, 2011] as proposed by Bernstein et al. [1987] (see Figure 2.1). In this distributed model, an arbitrary set of centralized DBMS are coupled together through a network in order to increase some of their individual features such as data availability (i.e., there are more servers to process requests) or fault tolerance (i.e., applications can connect to another server in case of failure) [Gray et al., 1996]. Nonetheless, the concurrency control manager is not local any more and, thus, transactions cannot be ordered (isolated and consistent) as easily as in the centralized scheme, which results in some extra overhead [Franaszek et al., 1992; Agrawal et al., 1994]. Note that from the application perspective, there is still no difference between either a centralized or a distributed layout since the DBMS is responsible for hiding its internal structure [Bernstein et al., 1987].

In fact, distributed databases [Abdelguerfi and Wong, 1998; Özsu and Valduriez, 1999] put then even more stress on the concurrency issues since (1) several applications can execute transactions at different sites (also referred to as nodes or servers), (2) every datum should keep its consistency (if replicated), and (3) transactions may contain operations accessing data from different servers [Lamport, 1978a]. In addition, this distributed architecture does not have an in-built mutual exclusion mechanism to ensure transactions' atomicity and ordering as in centralized schemes.

Therefore, distributed DBMSs implement their own techniques to cover the ACID properties and properly process transactions under concurrency situations [Bernstein and Goodman, 1981]. Specifically, there exist two major strategies to carry concurrency control in distributed databases [Thomasian, 1998a; Bernstein and Goodman, 1981]: (1) pessimistic concurrency control protocols and (2) optimistic concurrency control protocols.

1. Pessimistic concurrency control [Al-Jumah et al., 2000] assumes that transactions are potentially conflictive (i.e., different transactions access the same data items as shown in Example 2) and, thus, they have to wait for the end of all conflictive transactions under execution. Algorithms such as Strict Two Phase Locking (S2PL) keep data consistency and isolation using lock units over each data item in the database [Al-Jumah et al., 2000]. Then, each operation sends a lock request that is granted only if the lock is not being held by any other transaction. If the lock is not granted, the operation must—indefinitely—wait, which makes the strategy deadlock prone [Thomasian, 1998a,c; Özsu and Valduriez, 1999].

2. Optimistic concurrency control [Kung and Robinson, 1979] assumes that conflictive operations are unusual in the database and hence, each transaction is executed a priori and ordering issues are checked when it requests for its termination (i.e., at the `commit`/`abort` operation) [Thomas, 1979; Kung and Robinson, 1981]. Algorithms such as Optimistic Two Phase Locking (O2PL) [Carey and Livny, 1991] or Basic Optimistic Concurrency Control (BOCC) [Thomasian, 1998a,b] prevent operations from being blocked, which ensures system liveness. The key idea behind is that all operations are assumed to be compatible (i.e., non-conflictive) so they are executed when requested—and never delayed.

Certainly, the utilization of concurrency control algorithms [Franaszek et al., 1992] results in a considerable overhead due to the exchange of many network messages used to check the status of remote data items, which considerably slows down the transactions execution. This issue envisages some amazing research challenges—even in modern systems [Aguilera et al., 2009]—that attempt to minimize the number and size of these synchronization messages without violating the ACID properties.

In addition to concurrency control protocols, replicated databases also implement different (1) replication strategies [Wiesmann and Schiper, 2005]—set of policies to be applied when replicating data—which are chosen according to data access patterns and (2) group communication systems [Chockler et al., 2001]—set of messages and protocols to deal with fault recovery and ensure data synchronization between different servers—which are chosen according to the selected concurrency control protocol and replication policy. The global correctness criterion is used to theoretically validate that all transactions have been executed precisely [Bernstein et al., 1987]. These three ideas are elaborated in what follows.

2.3 REPLICATION TECHNIQUES

Data replication consists in placing the same object (also referred to as replica) on different sites [Özsu and Valduriez, 1999]. Database objects such as data items, rows, or tables can be replicated to provide better availability and fault tolerance [Bernstein et al., 1987; Lamport, 1998; Krikellas et al., 2010; van Steen and Pierre, 2010]. This has a strong impact on how transactions are processed and on the database features, especially scalability [Serrano et al., 2007], consistency [Kemme and Alonso, 2000a], isolation [Elnikety et al., 2005], durability [Aguilera et al., 2009], messages overhead [Wiesmann and Schiper, 2005], response time [Curino et al., 2010], and the global system throughput [Gray et al., 1996].

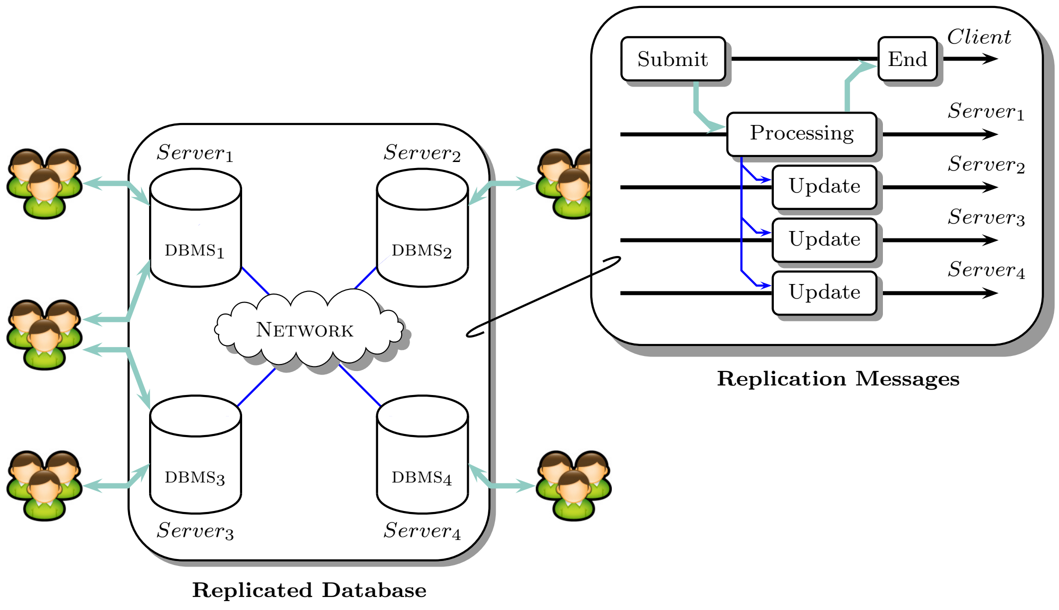


Figure 2.2: Replicated database model proposed by Wiesmann and Schiper [2005] and generic replication messages diagram. Clients issue transactions (green arrows) to a server, and it propagates them to the other ones (blue arrows), and finally it replies to the client.

As depicted in the left side of Figure 2.2 replication allows to keep a short physical distance between data and clients (i.e., moving data to computation), which shall improve the queries response time. However, as also depicted in the right side of Figure 2.2, each time a transaction is submitted to any server of the distributed database, a set of synchronization messages to update all other replicas' status is generated—in addition to the ones generated by the concurrency control protocol—, which may negatively affect the system performance [Gray et al., 1996]. Such message overhead comes from the fact that the replica manager—belonging to the transaction manager and behaving as a complement to the concurrency control—is responsible for keeping data consistency in the whole database [Bernstein et al., 1987]. This is commonly achieved [Jiménez-Peris et al., 2003] using either vote techniques—that assign some read and write quorum quotes to each object [Gifford, 1979]—or Read One Write All (ROWA) protocols—that perform read operations on a single site and write operations on all sites. The major difference between both strategies remains on how read operations are processed: voting protocols query a certain amount of replicas in order to get the most recent version of an item [Gifford, 1979; DeCandia et al., 2007] and ROWA assumes that all replicas have eventually converged and hence, queries a single replica. Actually, ROWA can be seen as a particular case of the voting protocol to minimize the number of messages associated to read operations.

Replication is one of the major causes of the stringent scalability in distributed databases [Gray et al., 1996]. Efforts have been made to explore this problem and propose different solutions to minimize the overhead associated to the replication process and, consequently, improve scalability [Muñoz-Escoí et al., 2007]. For instance, a trivial way to minimize the amount of exchanged messages consists in either reducing or limiting the number of servers that take part in the replication process (also referred to as partial replication). This approach improves the scalability but reduces the fault tolerance and availability [de Sousa et al., 2001; Serrano et al., 2007; Bernabé-Gisbert et al., 2008].

The literature presents several classifications for replication protocols [Gray et al., 1996; Wiesmann et al., 2000]. This thesis relies on the classification proposed by Wiesmann and Schiper [2005], which identifies five different techniques: (1) active/passive, (2) weak-voting, (3) certification-based, (4) update-everywhere, and (5) eager/lazy replication. The major differences among them are emphasized in the following:

Active/Passive replication were introduced by Lamport [1978b] and Alsberg and Day [1976] respectively. As depicted in Figure 2.3, active replication (also referred to as state machine replication) processes the same request at all replicas (i.e., the whole transaction is broadcasted to all sites) [Amir and Tutu, 2002]. To ensure that all replicas reach the same outcome and keep data consistency, active replication requires that (1) transaction execution has to be deterministic at every server, (2) all operations in the transaction have to be known in advance (no interactive transactions are allowed), and (3) a reliable communication service (i.e., total order broadcast) has to guarantee that transactions arrive in the same order to all replicas [van Renesse and Guerraoui, 2010].

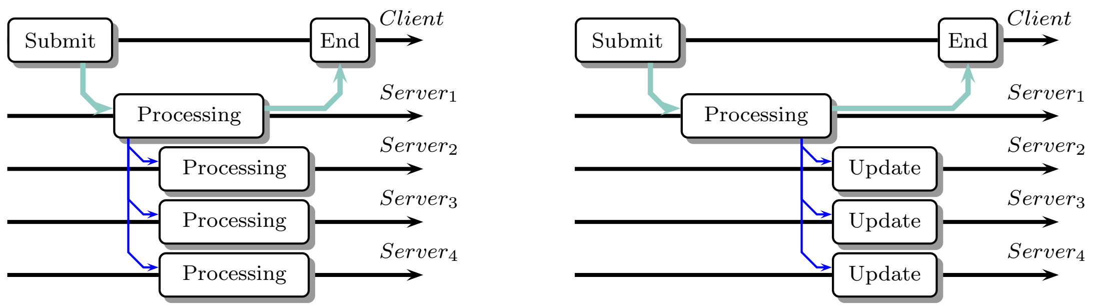


Figure 2.3: In active replication (left diagram), all replicas process the same operations. In passive replication (right diagram), delegate replica processes operations and transfers the resulting state to other replicas.

On the contrary, in passive replication (also referred to as primary-copy or primary-backup) a single master site processes all transactions and transfers the resulting state to the other replicas or secondaries that act as backups (see Figure 2.3). This approach can get easily overloaded and has limited scalability. However, in this case interactive transactions are supported and no deterministic behavior is required in all replicas since they only receive the resulting updates of the transaction [Défago and Schiper, 2004]. Nonetheless, to satisfy the ordering requirements, a group communication service is still needed to ensure that all remote updates are correctly applied.

Active replication is usually selected on failure prone scenarios because it can mask failures without performance degradation. Passive replication is chosen when transactions require high computing resources (i.e., a single site performs heavy calculations and broadcasts the obtained results to all replicas) [van Renesse and Guerraoui, 2010]. Hence, it is important to note that the performance in active replication is considerably degraded due to the fact that all replicas have to perform the same work [Wiesmann and Schiper, 2005].

Weak-voting replication, introduced by Kemme and Alonso [2000b], is inspired by the replica control management based on voting techniques [Gifford, 1979]. This strategy minimizes the overhead of the replication protocol by (1) reducing the number and size of messages used to synchronize replicas, (2) eliminating deadlocks (i.e., transactions that wait for the termination of other transactions), and (3) reducing consistency [Fekete and Ramamritham, 2010; Bernstein and Das, 2013].

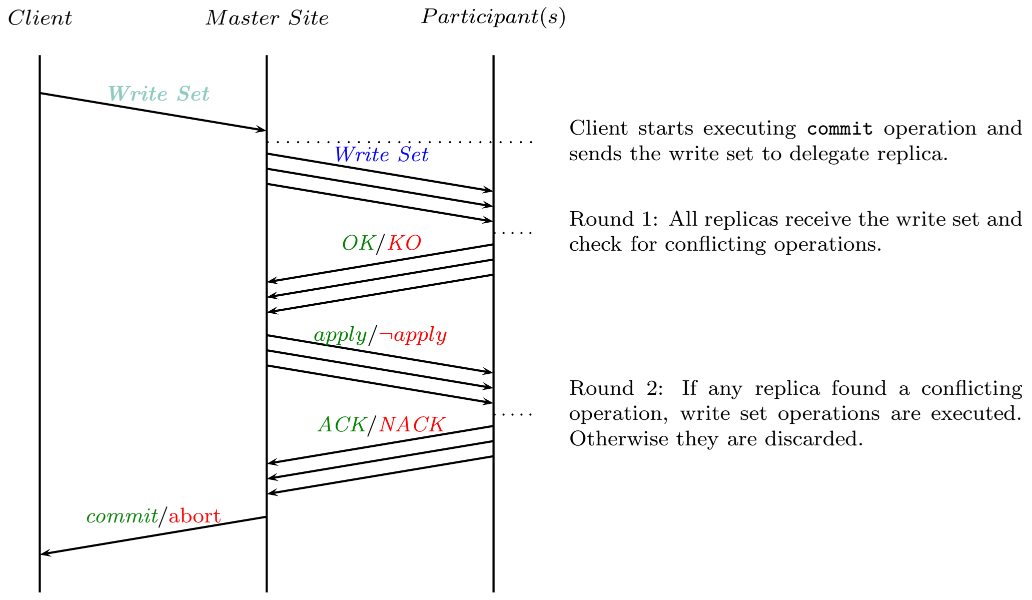

Client
Master Site
Participant(s)
Write Set
Write Set
Client starts executing commit operation and sends the write set to delegate replica.
OK/KO
Round 1: All replicas receive the write set and check for conflicting operations.
apply/¬apply
ACK/NACK
Round 2: If any replica found a conflicting operation, write set operations are executed. Otherwise they are discarded.
commit/abort

Figure 2.4: Two-phase commit protocol. The write set is forwarded to all replicas, which individually check for possible conflicts. If no conflicts are detected, operations in the write set are executed. Otherwise, they are discarded.

This strategy (1) implements the functions of the concurrency control manager using a consensus protocol (i.e., 2-Phase Commit (2PC) [Pedone et al., 1998; Kemme and Alonso, 1998]) in the replication process [Défago and Schiper, 2004], and (2) uses a hybrid approach between passive and active replication to update remote replicas. Specifically, weak-voting replication strategies implement the following procedure:

1. Execute the transaction at a delegate replica.
2. Broadcast the write operations of the transaction (also referred to as Write Set (WS)) to all replicas.
3. Ask to all replicas if they can execute the WS without colliding with their local transactions (see first round of the concurrency control messages in Figure 2.4).
4. Give the order to either (1) execute the WS if all replicas agreed on not having conflicts, or (2) to not execute the WS if any replica detected a conflict (see second round of the concurrency control messages in Figure 2.4).
5. Notify the client if the transaction has been successfully executed (i.e., commit or abort).

This technique improves the performance of passive replication in the sense that read operations (also referred to as Read Set (RS)) are not broadcasted to all replicas and, thus, read-only transactions—very common in several applications such as mail lists, DNS servers, or web sites—can be fully executed in the delegate site. This minimizes the amount of synchronization messages and, as a consequence, the overhead. Unfortunatelly, (1) it adds an extra round of synchronization messages (due to the 2PC protocol), and (2) still demands for a reliable group communication service to properly order the messages on broadcast (at least for the first round) [Muñoz-Escoí et al., 2007].

Certification-based replication was introduced by Pedone [1999] and was aimed to reduce the overhead associated to the second round of 2PC messages in weak-voting replication protocols. To achieve such a goal, it implements an—optimistic—concurrency control inside the replication protocol that is combined with a reliable group communication service (i.e., atomic broadcast) [Pedone et al., 1998]. In this way, a deterministic verification process is used to validate transactions, which allows each replica to decide by itself if the transaction is conflictive without further synchronization messages.

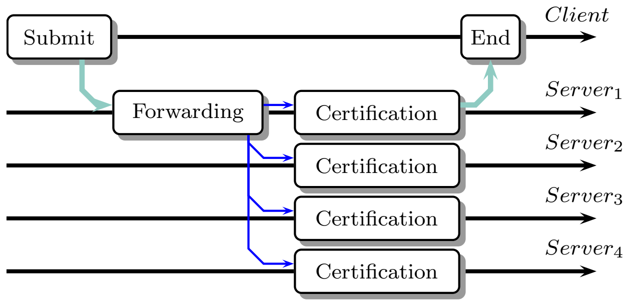


Figure 2.5: Certification-based replication protocol. All replicas execute a determinist algorithm to assert whether operations are conflictive.

Therefore, the whole transaction is optimistically executed at a delegate site (as in passive replication) but delaying the write operations. Once the client issues the `commit` operation, the delegate replica broadcasts the RS and WS to all replicas using the group communication service (blue arrows in Figure 2.5)—which ensures that all messages will be delivered in the same order. As all nodes run the same concurrency policy (i.e., certification) in order to process the transaction, the delegate replica does not need any further messages nor vote rounds to notify the client if the transaction has been committed [Kemme and Alonso, 2000b; Wiesmann and Schiper, 2005; Muñoz-Escoí et al., 2007].

This approach minimizes the number of exchanged messages between replicas but increases their length (remind that the RS is also included in the diffusion process). However, there may exist some situations where this RS does not need to be broadcasted if further considerations are assumed (e.g., reducing the transactions isolation level) [Elnikety et al., 2005; Lin et al., 2005; Adya et al., 2000].

Update-everywhere replication can be seen as a generalization of the active replication introduced by Lamport [1978b]. This strategy alleviates the bottleneck effect originated in those systems that use a centralized control of transactions (e.g., active and passive replication) by allowing the execution of any update operation at any replica and, thus, improve the fault tolerance. In addition, update-everywhere replication does not explicitly require a deterministic behavior at each replica when executing operations; for instance the replica agreement and coordination process might be carried out using 2PC [Pedone et al., 2000] (see Figure 2.4). Note that the recovery process in case of failure is still efficient because all replicas contain the same state, which allows the client to be effectively redirected.

Eager/Lazy replication were introduced by Bernstein and Goodman [1983] and Ladin et al. [1991] respectively. These strategies are targeted to improve the transactions response time by reducing the consistency facilities of the database.

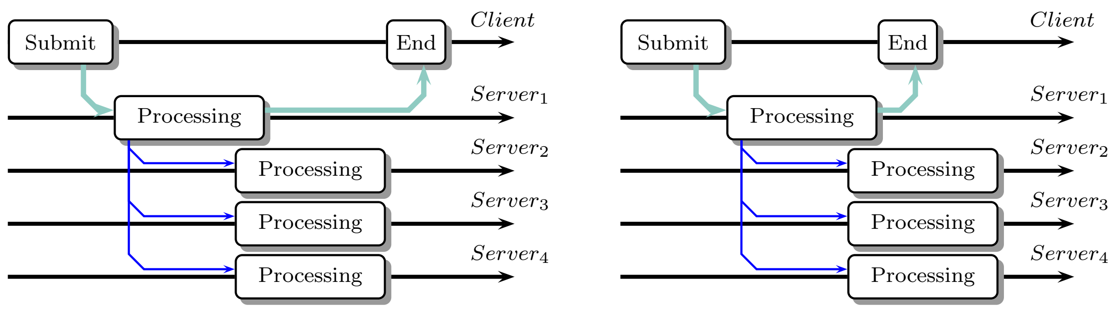


Figure 2.6: In eager replication (left diagram), delegate replica waits for all replicas to have applied the transaction. In lazy replication (right diagram), delegate replica replies to client as soon as it has finished processing the transaction.

On the one hand, eager replication (also referred to as synchronous replication) [Bernstein et al., 1987] waits for all replicas to have executed the whole transaction before notifying the client about the transaction outcome (i.e., at commit/abort operation), which ensures strong consistency [Adya, 1999; Vogels, 2009; Bernstein and Das, 2013] but has limited scalability—the more replicas to be updated, the more messages to be sent and the more time the synchronization process takes (see Figure 2.6).

On the other hand, lazy replication (also referred to as optimistic or asynchronous replication) [Krikellas et al., 2010; Wiesmann and Schiper, 2005] notifies the client about the transaction outcome before all replicas are synchronized, which may reduce the database consistency degree (also referred to as weak consistency [Adya, 1999; Vogels, 2009]) because replicas may diverge, but improves its scalability and response time (see Figure 2.6).

Despite this classification, there exist hybrid solutions such as eager primary-backup or lazy primary-backup that combine the benefits of each technique and adapt to each application demands [Wiesmann et al., 2000; Pedone et al., 2000]. Nevertheless, all these combinations should never contradict Brewer's CAP theorem [Brewer, 2000; Gilbert and Lynch, 2002; Brewer, 2012], which states that it is not possible to have a scalable distributed database (1) strongly consistent (e.g., provided by eager replication), (2) highly available (e.g., provided by update-everywhere replication), and (3) fully network-partition tolerant (e.g., provided by weak-voting replication). Therefore, designers have to carefully choose which features can be relaxed and select the most suitable replication strategy.

In addition, database architects have to keep in mind that several replication protocols—apart from implementing some of the concurrency control features (e.g., 2PC, certification)—demand for a reliable group communication service in order to guarantee that messages are properly delivered to each replica, which also adds an extra overhead to the database communications. The following subsection expands this idea and reviews existing group communication systems to fulfill replication protocols demands.

2.4 GROUP COMMUNICATION SYSTEMS

Fault tolerance jointly to network communications—derived from the unavoidable concurrency demands—suppose one of the major design challenges [Défago et al., 2004] in the distributed systems field (e.g., a delegate replica may reach a deadlock state if a communication link is faulty or the global consistency may be violated if synchronization messages are not delivered in the same order to all sites). While point-to-point communications are widely solved by lower layers in the Open Systems Interconnection (OSI) reference model (i.e., Transport Control Protocol (TCP)), complex multi-point-to-multi-point dialogs are due to be managed at the application level.

Therefore, Group Communication Systems (GCSs) provide a set of reliable tools and services in form of primitives committed to facilitate fault-tolerant and multi-point to multi-point connections between groups of distributed processes [Chockler et al., 2001] (e.g., replicas applying the WS during the replication process). According to each application constraints and goals, several GCS specifications have been presented [Schiper and Raynal, 1996; Pedone, 1999; Chockler et al., 2001; Défago et al., 2004; Schiper, 2006] leading to a large number of solutions to be chosen by designers. This section reviews the most relevant features (also referred to as services) of the GCS that actually complement the task of the replication and concurrency control protocols: membership service, communication service, and virtual synchrony [Arrieta-Salinas, 2012].

Membership service notifies which processes are available within a given group (also referred to as view). As processes running in fault tolerant distributed systems can join or leave at will, it is necessary to provide them with a reliable view—logical representation of the group membership—of the active and connected processes in the whole system [Chockler et al., 2001].

This service was first introduced by Birman and Joseph [1987] as a solution for the design of a distributed computer system with support for fault-tolerant process groups. They designed a family of reliable multicast protocols that attained high levels of concurrency while respecting application-specific delivery ordering constraints. In fact, they defined two different types of service according to the group composition:

1. Primary partition service. It maintains the same group membership perception at all processes since views to be installed are totally ordered (i.e., delivered everywhere in the same order).

2. Partitionable service. It does not ensure the same group membership perception at all processes since views to be installed are partially ordered (i.e., multiple disjoint and divergent views may coexist concurrently).

Regardless the significant overhead in terms of network messages added by this service—mostly due to the implementation of the message delivery service and views diffusion—it is only aimed at detecting which processes are alive and connected in the distributed system. The following module of the GCS, coined as communication service, offers a set of primitives to allow reliable application dialog between the aforementioned processes.

Communication service provides a multicast communication toolset (i.e., a set of primitives to allow multi-point-to-multi-point dialogs) to connect processes inside a view—previously provided by the membership service. Thus, it (1) specifies the guarantees concerning messages delivery (also referred to as reliability) and (2) restricts the order in which messages are delivered (also referred to as ordering properties [Hadzilacos and Toueg, 1994]).

According to each application demands, the GCS can provide three different reliability degrees for multicast messages through its communication service [Bartoli, 2004]:

1. Unreliable multicast. It provides the lowest reliability degree since neither message losses nor drops are prevented.
2. Reliable multicast. It provides a higher reliability degree because it ensures that a multicast message is delivered to all intended receivers that do not crash. It satisfies the following three basic properties [Rodrigues and Raynal, 2000; Chockler et al., 2001; Défago et al., 2004; Birman, 2012]:
   a) Validity. If a correct process multicasts a message m, then all correct processes will eventually receive m.
   b) Agreement. If a correct process delivers a message m in view V, then all correct processes of V will eventually deliver m.
   c) Integrity. For any message m, every correct process will deliver m at most once.
3. Uniform reliable multicast. It provides the highest reliability degree because it ensures that a message that is delivered by a member (even if it fails), will be delivered at all available members.

   To provide such a guarantee the GCS must ensure that all active members in the current view have received a message prior sending it to a process, which considerably increases the system overhead in terms of network messages. Uniform reliable multicast can be specified and better understood by requiring the following two properties [Hadzilacos and Toueg, 1994; Arrieta-Salinas, 2012]:
   a) Uniform agreement. If a process—whether correct or faulty—delivers a message m in view V, then all correct processes of V will eventually deliver m.
   b) Uniform integrity. For any message m, every process will deliver m at most once, and only if m is previously multicast by a process.

Such a variety of reliability degrees becomes useful in replicated environments such as distributed databases; designers can adapt the GCS to the nature of the processes and their failure model. For instance, uniform reliable multicast allows sites to broadcast an update, ensuring that every other replica in the current view will eventually either apply it or crash (i.e., avoiding false updates).

In addition to the reliability degree, the communication service also provides different guarantees concerning the order in which messages are delivered to processes. According to [Hadzilacos and Toueg, 1994; Bartoli, 2004], the most common ordering guarantees are:

1. First In First Out (FIFO) order. It ensures that messages that have been multicast by a given process are delivered according to the order in which they have been sent (e.g., TCP).
2. Causal order. It guarantees (1) FIFO order and (2) that a reply in response of a multicast message m will be always delivered after the delivery of m.
3. Total order. It guarantees that all members in a view deliver messages in the same order irrespective of which process multicasts them.

   Note that although total order does implicitly include neither FIFO order nor causal order, it can be combined to obtain FIFO total order or causal total order guarantees [Défago et al., 2004].

Certainly, combining uniform reliable multicast with total order guarantees makes the design of replication and concurrency protocols easier. In this case, it is possible to assume that the communication channel is deterministic due to the guarantees provided by the uniform multicast.

Despite the amazing guarantees provided by the membership and communication services, distributed databases usually require an extra degree of synchronization for all updates to be applied to the same view, which leads to the definition of virtual synchrony described in what follows.

Virtual synchrony is a property aimed to guarantee that all messages multicast in a given view are already delivered before installing a newer view. Formally:

If two processes p and q install the same view V over the same previous view V', then any message received by p in V' is also received by q in V' [Chockler et al., 2001].

Therefore, view changes are seen as synchronization points in the sense that multicast messages are ordered with respect to view changes: processes that install view V in view V' have all received the set of multicast messages <M> originated between the two views [Arrieta-Salinas, 2012]. This property (also referred to as failure atomicity or message agreement) was first introduced by Birman and Joseph [1987] in the context of a primary partition membership service and later extended to a partitionable service by Friedman and van Renesse [1996].

Virtual synchrony is very useful when implementing state-machine replication, which uses total order multicast messages to keep consistency between replicas. In this case, when a replica becomes disconnected—due to either communication link failure or machine fault—it can reach different and divergent states because it cannot receive synchronization messages. However, if virtual synchrony is assured (and properly optimized to minimize the messages overhead [Amir et al., 1997])

(1) this situation may be avoided due to the fact that updates will not be applied to out-of-date views, and (2) state transfer among processes that continue together from one view to another will be avoided as well.

Overall, GCS strategies provide a suitable component to develop a reliable transactional replicated database. However, due to the wide amount of protocols and techniques that may fit and successfully run in this context, it is necessary to verify that ACID properties are never violated. Thus, a set of correctness criteria were defined by Bernstein et al. [1987].

2.5 correctness criteria

Besides storage and retrieval, DBMSs provide a reliable framework able to hold read and write operations over data under high concurrency scenarios. Therefore, data are replicated—using the replication protocols and GCS described in Section 2.3 and Section 2.4—over an arbitrary set of servers to both increase data availability and allow different applications run simultaneously [Gray et al., 1996]. However, concurrent operations issued over replicated data incur in the risk of potentially violating consistency and, thus, DBMS correctness [Bernstein et al., 1987]. This subsection (1) states the correctness concept in the context of a distributed DBMS and (2) describes the two best known criteria to check it: 1-Copy Serializability (1SR) and 1-Copy Multi-Version (1CMV).

The most general and restrictive definition of DBMS correctness claims that ACID properties can never be violated [Bernstein et al., 1987]. However, these properties are often difficult to quantify, which drove Bernstein et al. [1987] to state a more accurate definition for correctness criterion in a DBMS: A given distributed DBMS is correct if it satisfies the following two constraints:

1. The state of the database at the end of a set of concurrent transactions is the same as the one resulting from some serial execution (i.e., transactions are executed non-concurrently).

2. Transactions' outcome when executed concurrently is the same that if they were executed in the aforesaid serial order.

These restrictions [Ramamritham and Chrysanthis, 1996] impose that correct distributed databases—irrespective of their concurrency control protocol, replication strategy, and GCS used—must behave as a single site (also referred to as 1-Copy) from the application point of view. As there are several applications that can implement relaxed forms of the ACID properties [Adya, 1999; Berenson et al., 2007; Vogels, 2009; Aguilera et al., 2009; Jones et al., 2010; Bernstein and Das, 2013] due to their intrinsic nature (e.g., web servers, mail lists), two different correctness criteria were defined [Bernstein et al., 1987]:

1-Copy Serializability (1SR) is one of the strongest correctness criterion for a DBMS and is achieved when the previous two constraints are fulfilled. Basically, it assumes that if all transactions are correct separately, their serial execution, providing complete isolation between them, will also be correct.

This correctness criterion comes from the implementation of the strongest isolation level (also referred to as serializable [Bernstein et al., 1987]) on the DBMS, which demands that changes produced by a transaction have to become visible before beginning another transaction. However, providing such isolation level (1) is very expensive—in terms of network messages—for the concurrency control and replication protocols, and (2) too restrictive for many real-world applications [Ramamritham and Chrysanthis, 1996].

1-Copy Multi-Version (1CMV) is a weaker correctness criterion for a DBMS and is achieved when the ordering properties of the ACID transactions (i.e., isolation and consistency) are relaxed [Bernstein and Das, 2013]. Basically, it assumes that each `write` operation over a data item x produces a new copy (also referred to as version) of x [Bernstein et al., 1987]. Thus, transactions have the freedom to access any combination of versions stored in the database when this criterion is adopted.

Therefore, there exist several correctness degrees for 1CMV according to the isolation level (i.e., the range of object versions available for a transaction) implemented by the DBMS [Berenson et al., 2007]:

- **1-Copy Snapshot Isolation (1CSI)** is the strongest 1CMV correctness criterion. It was defined to forestall the typical concurrency issues found in 1CMV criterion, specifically *phantom reads*, *non-repeatable reads*, and *dirty reads*. To achieve such a goal, it prevents each transaction from accessing different versions of the same object—as other 1CMV criteria do. This correctness criterion comes from the implementation of the snapshot isolation level [Lin et al., 2005; Daudjee and Salem, 2006; Bernabé-Gisbert et al., 2008; Lin et al., 2009], which forces transactions to work with the same version (also referred to as snapshot) of all selected data items until the end of the transaction.

  Hence, 1CSI states that (1) read operations never block between themselves—they might belong to different consistent snapshots—and (2) write operations never block read operations—they will be applied to another snapshot.

  As a result, this correctness criterion provides the same guarantees than 1SR in terms of giving a read consistent view [Bernstein et al., 1987] of the database and it considerably improves its performance [Elnikety et al., 2005; Serrano et al., 2007]. Such improvement comes from a subtle difference between both isolation levels (and, thus, their respective correctness criteria): while serializable isolation avoids writing to an object which is being concurrently read (i.e., aborts conflictive transactions), snapshot isolation hides these updates by applying them to a newer version that will be never observed by the transaction which is currently reading that item and, hence, allow *conflictive* transactions progress subsequently.

- **1C-Repeatable-Reads** is a weaker 1CMV correctness criterion. It comes from the implementation of the repeatable read isolation level [Bernstein et al., 1987], which keeps `write` and `read` locks over selected data items until the end of the transaction.

  As in repeatable read isolation level locks are only applied to existing data, this correctness criterion is exposed to the *phantom reads* phenomena [Bernstein et al., 1987]: if a transaction performs a query over the same data range (i.e., retrieving a set of objects meeting a given condition) multiple times, it may get different collections of rows for each query if another concurrent transaction is adding new objects, as shown in Example 3.

Example 3. *This example illustrates two transactions* T_0 *and* T_1*:* T_0 *reads twice from all objects* X *meeting a condition* Y*,* T_1 *inserts more objects* X *meeting the same condition* Y*, and finally both transactions commit. Next, it exhibits a possible history which leads to the phantom read phenomena.*

$$T_0 = \{b_0, r_0(X|Y), r_0(X|Y), c_0\},$$
$$T_1 = \{b_1, w_1(X|Y), c_1\},$$

$$T_0 \cup T_1 = \{b_0, b_1, r_0(X|Y), w_1(X|Y), c_1, r_0(X|Y), c_0\}.$$

The second read operation obtains more rows than the first read due to the execution of T_1*.*

Therefore, this correctness criterion might be suitable for environments where data is loaded once onto the database and no more rows are added over time.

- **1C-Read-Committed** is a weaker 1CMV correctness criterion. It comes from the implementation of the read committed isolation level [Bernstein et al., 1987], which only keeps `write` locks over selected data items until the end of the transaction.

  Apart from inheriting the weaknesses of the 1C-Repeatable-Reads (i.e., *phantom reads*), this correctness criterion is also exposed to the *non-repeatable reads* phenomena [Bernstein et al., 1987]: if a transaction performs the same query (i.e., retrieving the value of a given object) multiple times, it may get different values if another concurrent transaction is updating that objects, as shown in Example 4.

  Example 4. *This example illustrates two transactions* T_0 *and* T_1*:* T_0 *reads twice from object* X*,* T_1 *updates object* X*, and finally both transactions commit. Next, it exhibits a possible history which leads to the non-repeatable read phenomena.*

  $$T_0 = \{b_0, r_0(X), r_0(X), c_0\},$$
  $$T_1 = \{b_1, w_1(X), c_1\},$$

  $$T_0 \cup T_1 = \{b_0, b_1, r_0(X), w_1(X), c_1, r_0(X), c_0\}.$$

  The second read operation obtains a different value for object X *than the first read due to the execution of* T_1*.*

  Note that versions of each data item can be accessed by transactions without further restrictions, which leads to the two aforementioned issues. This correctness criterion might be suitable for environments where updates are infrequent such as DNS or web servers.

- **1C-Read-Uncommitted** is the weakest 1CMV correctness criterion. It comes from the implementation of the read uncommitted isolation level [Bernstein et al., 1987], which keeps no locks over selected data items. Thus, transactions may work with changes made by other transactions that are not already committed.

  Apart from inheriting the weaknesses of the 1C-Read-Committed—and thus, 1C-Repeatable-Reads—(i.e., *non-repeatable reads* and *phantom reads*), this correctness criterion is also exposed to the *dirty reads* phenomena [Bernstein et al., 1987]: if a transaction performs a query over an object which has been previously updated by another transaction that will abort, it might get a value for that object which should have never been available in the database, as shown in Example 5.

Example 5. *This example illustrates two transactions* T_0 *and* T_1*:* T_0 *updates object* X*,* T_1 *reads object* X*, and finally* T_0 *aborts and* T_1 *commits. Next, it exhibits a possible history that leads to the dirty read.*

$$T_0 = \{b_0, w_0(X), a_0\},$$
$$T_1 = \{b_1, r_1(X), c_1\},$$

$$T_0 \cup T_1 = \{b_0, b_1, w_0(X), r_1(X), c_1, a_0\}.$$

The read operation obtains an invalid value for object X *due to the fact that* T_0 *has aborted.*

Therefore, database designers are in charge of selecting the most suitable correctness criterion in order to assure that (1) concurrency control manager, (2) replication manager, and (3) GCS effectively keep ACID properties according to each application nature.

However, there are some situations where the distributed DBMS correctness is completed by a (small) part of the application layer (also referred to as middleware) [Lin et al., 2005; Patiño-Martínez et al., 2005; Muñoz-Escoí et al., 2006]. That is, the database does not provide enough guarantees by its own and thus, needs a software layer to avoid *malicious* actions. This strategy increases the flexibility of the global system (i.e., practitioners can develop specialized solutions) at detriment of the development cost (i.e., the distributed database does not behave as a single black box anymore). Actually, designers can develop their own specification for replication, concurrency, and/or GCS.

Nevertheless, sometimes it is complicated to proof that a particular specification fulfills a given correctness criterion. Therefore, the remainder of this chapter is devoted to propose the design and development of a simulation environment for a generic distributed DBMS. This framework will simplify the protocols prototyping and the verification of their correctness. In the next section a distributed DBMS model that combines the conceptual model proposed by Bernstein et al. [1987] and the one proposed by Wiesmann and Schiper [2005] is introduced.

2.6 ABSTRACT MODEL FOR DISTRIBUTED DATABASES

Distributed databases and their internal modules have been widely modeled in the past. So far, practitioners have presented centralized and distributed conceptual layouts [Bernstein et al., 1987], database concurrent access patterns [Thomasian, 1998b], replication management prototypes [Wiesmann and Schiper, 2005], and distributed systems analytical behavior models [Serrano et al., 2007]. This section consolidates all these ideas into a single, generic proposal that combines all the previous approaches. This will set the guidelines and fundamentals to develop a simulation environment based on the Parallelism and Abstraction Distributed Design – the RAmon Llull Environment (PADD/RALE) Integrated Development Environment (IDE) [Babot, 2009; Beltran, 2010] detailed in the following chapter.

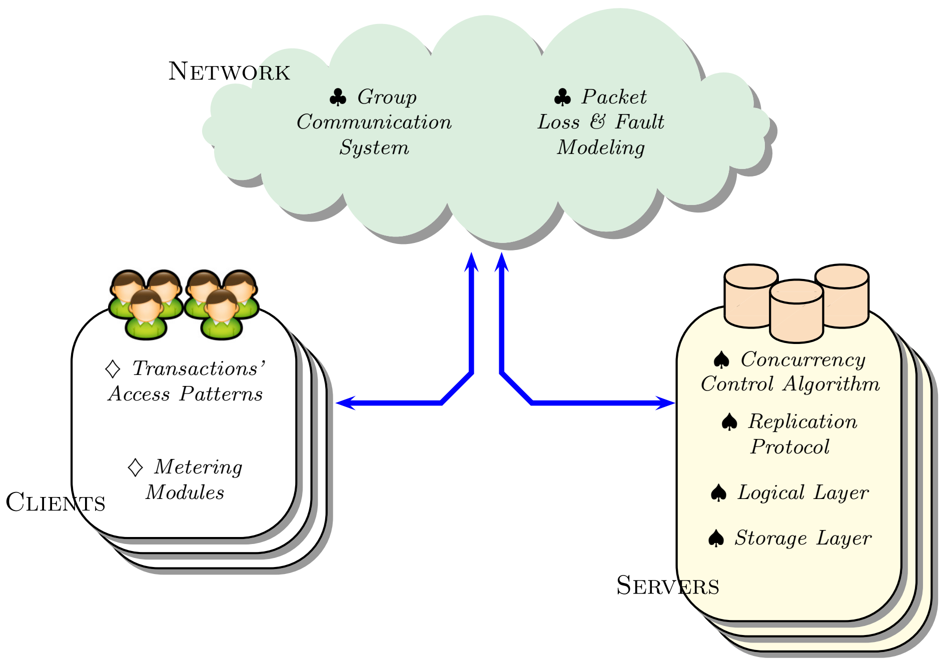

NETWORK
♣ Group Communication System
♣ Packet Loss & Fault Modeling
◇ Transactions' Access Patterns
◇ Metering Modules
CLIENTS
♠ Concurrency Control Algorithm
♠ Replication Protocol
♠ Logical Layer
♠ Storage Layer
SERVERS

Figure 2.7: Distributed database abstract model. The database is split into three modules: clients, network, and server(s).

As depicted in Figure 2.7, we propose to split the database—no matter distributed or not—into three independent modules: clients/applications, network, and server(s). Each one is described in what follows:

Servers. This module implements all functionalities concerning data storage and retrieval. Specifically, it contains the following blocks:

- Concurrency control algorithm. It ensures safe concurrent data accesses according to the policy determined by the specified concurrency control algorithm.
- Replication protocol. It implements all the directives regarding the replication process. This block is also in charge of hiding the internal layout of the database to clients and applications (i.e., each server—also referred to as delegate replica—behaves as a proxy of any application instance).
- Logical layer. It keeps all data schemas, views, and dictionaries up to date (i.e., when an item is added or removed from a replica, the data dictionary [Bernstein et al., 1987] of all servers is updated accordingly).
- Storage layer. It models the physical storage layer: hard disk(s), main memory, etc.

Note that multiple instances of this module can exist; each one holds its own physical characteristics (e.g., disk input/output access time, memory size, or processing capacity).

Network. This module models the behavior of a general-purpose network that delivers packets to servers and clients [Tanenbaum and Steen, 2006; Birman, 2012]. Actually, the goal of the network module is manyfold: (1) model any network configuration and topology (e.g., meshed, star, ring), (2) model network protocols overhead, (3) model network issues (e.g., congestion, packet delay, packet loss), and (4) model GCS [Hadzilacos and Toueg, 1994] used by the server's module. To this end, the module is decomposed into the following blocks:

- Group communication system. It implements the packet delivery policies (see Section 2.4) according to the selected GCS [Défago et al., 2004] and configured network characteristics.
- Packet loss & fault modeling. It implements the packet loss probability distribution function and faulty links policy (e.g., a server becomes unavailable during a certain period).

Clients/Applications. This module models the lowest layer of the application side, which is issuing transactions to the database and receiving their outcomes. In addition, it computes the throughput of the database according to the submitted load [Nambiar et al., 2010]. Specifically, it contains the following blocks:

- Transactions' access patterns. It models the type of transactions (e.g., read-only, update-intensive) submitted to the database and their data access patterns (e.g., hot-spot area size, hot-spot area hit frequency, operation size) [Kemme, 2000].
- Metering module. It assesses the performance of the database by measuring some metrics of interest (e.g., transaction time, restart ratio, throughput, abort rate) [Kemme, 2000].

Again, note that multiple instances of this module can exist; each one holds its own specific characteristics (e.g., transaction type, or issued load) and results. As each client collects a partial view of these results, a data aggregation process has to be conducted in order to unify them.

In this section, we have placed the previously described database and replication concepts into an abstract database model. This abstraction constitutes the core layer of the simulator for distributed transactional databases proposed in the following chapter.

Contribution.

1. Revision of transaction management principles.
2. Enumeration of the most relevant replication techniques in transactional distributed databases.
3. Overview of existing GCSs and properties that ease transaction processing in replicated databases.
4. Revision of correctness criteria that validate distributed databases behavior.
5. Abstraction and decomposition of a distributed database into three logical modules.

3 PERFORMANCE ANALYSIS OF CONCURRENCY CONTROL AND REPLICATION PROTOCOLS

Summary. Including innovative improvements to existing concurrency control and replication protocols—or even developing new ones—and reliably comparing their performance is a time-consuming and challenging task. This chapter presents a simulation environment that enables practitioners to quickly deploy new advances in transactional distributed databases and accurately analyze the effects of their achievements. Conducted experiments with this simulator portrait the scalability limitations of classic protocols and how their performance rockets when transactions size and scope is reduced[2].

"Those who cannot remember the past are condemned to repeat it"
— George Santayana , 1905.

3.1 INTRODUCTION

Distributed systems, specially large replicated databases, are complex to manage and deploy: budget constraints, system configuration, erratic user demands, unexpected faults, operating system secondary tasks, unexpected network traffic, etc. This situation drives practitioners into time consuming experimentations and hard to analyze results. Therefore, several proposals in the literature avoid dealing with such issues by either (1) proposing analytical models and validating them in small scenarios [Serrano et al., 2007], (2) using middleware-based frameworks that somehow abstract the physical layer of the distributed system and focus on particular database functionalities (e.g., partitioning, replication, fault tolerance, scalability) [Lin et al., 2005; Patiño-Martínez et al., 2005; Muñoz-Escoí et al., 2006; Plattner, 2006], (3) conducting tedious formal reasonings that proof the correctness of a given proposal [Kemme, 2000], or (4) using simulation-based environments to accurately emulate some of the distributed system specificities [Zaballos et al., 2010; Casteigts, 2010; Khan et al., 2011]. Typically, none of these approaches addresses the distributed system as a whole and, thus, some strong assumptions are asserted (e.g., predefined failure patterns, uniform network behavior, hardware utilization static models), which may pervert the obtained experimental results. This chapter proposes a comprehensive approach that allows a rapid prototyping of any module in a transactional database through a controlled simulation environment, which has been erected upon the abstract model detailed in the previous chapter.

2 An earlier abridged version of the work reported in this chapter was published as the paper entitled "Optimistic concurrency control with partial replication design" in the proceedings of the 2009 International Conference on Applied Computing (IADIS 2009).

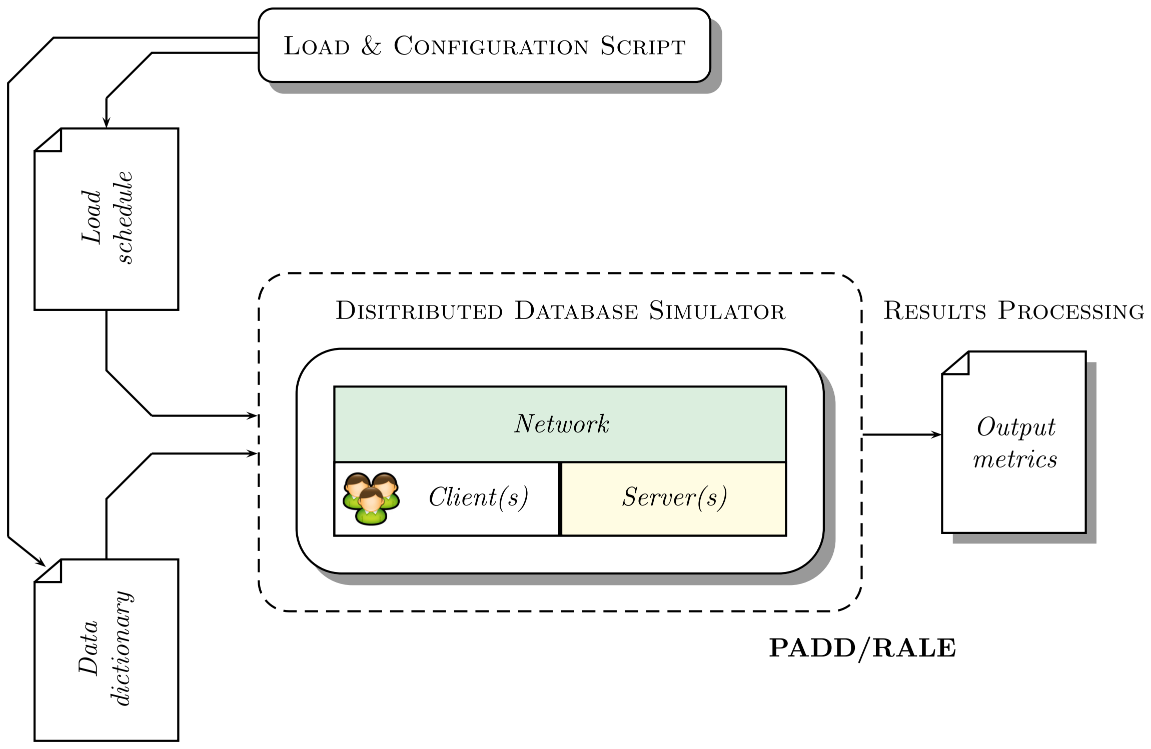


Figure 3.1: Simulator general layout.

Specifically, we first show the mapping between the abstract model and the three software modules that compose the simulator, then describe how traditional concurrency control and replication protocols have been implemented on this platform, and later present a performance evaluation of them when tested under different scenarios. The conclusions extracted from this analysis will support the fundamentals of the cloud storage paradigm described in Chapter 4.

3.2 SIMULATION ENVIRONMENT

Typically, distributed database designers develop and prototype their proposals either under (nearly) real-world frameworks or simulated environments.

Real-world scenarios are often complex to freeze at a certain point and some key system metrics may remain hidden or inaccessible due to physical constraints. For instance, if a system experiences an unusual response time from a network router at some specific time during an experiment, it might be because buffer overflows, unexpected packet sizes, massive errors at frames checksum, or internal processor overhead. Not being able to precisely identify the source of the issue prevents from rapidly understanding the outcome of the conducted experiments, and lead practitioners to perform time consuming trial and error tests to obtain conclusive results.

On the contrary, simulated environments are characterized by the simplicity and freedom to measure any parameter of the system without altering its natural behavior.

This section proposes a simulation environment to model a complete transactional distributed database—from the user-loads to the network behavior, including the concurrency control and replication protocols. In what follows, we describe the general

Algorithm 3.1 Load and configuration script behavior.

```
Require: $\exists$ a solution that satisfies all inputArguments[ ]
Result: Schedule file, Data dictionary file
 1: parse inputArguments
 2: numOperations := estimateOperationsAmount(transPerSecond, transPattern,
    numClients, numServers)
 3: repeat
 4:     operationSet := buildTrans(hotSpotSize, hotSpotFreq, transPattern)
 5:     updateDataDictionaryFile(operationSet, repDegree, numServers)
 6:     assign operationSet to client
 7:     updateScheduleFile(operationSet)
 8:     numOperations := numOperations − operationSet.size( )
 9: until numOperations $\leqslant 0$
10: shuffle $\forall$ operations in scheduleFile
```

layout of the simulator and its underlying technology, detail how each entity has been implemented, and provide a set of guidelines to measure its performance.

3.2.1 *General layout*

The simulation environment is composed of three separate entities, which are depicted in Figure 3.1: the load and configuration script, the database simulator, and the results processing module. Each module is detailed in what follows.

Load and configuration script. It is responsible for (1) configuring the simulator according to the constraints stated by each experiment, (2) building an object set to construct transactions (also referred to as workload generation), and (3) make an execution schedule for the database. Specifically, it builds a set of randomly-generated mock objects and requests that fulfill a set of constraints specified by the user. These constraints are defined to enable an eventual comparison between the outcome provided by the proposed simulator and results presented by other authors [Thomasian, 1998a; Al-Jumah et al., 2000; Kemme, 2000; Serrano et al., 2007; Florescu and Kossmann, 2009; Carstoiu et al., 2010; Nambiar et al., 2010; Shafer et al., 2010; van Steen and Pierre, 2010]:

- Number of servers. Amount of servers (also referred to as nodes) the database owns. For sake of simplicity, it is assumed that all nodes have the same features.
- Number of clients. Amount of clients that connect to each server. For sake of simplicity, it is assumed that all servers handle the same number of clients.
- Number of transactions per second. Amount of transactions that a given client issues against the database. Note that these transactions are considered to be stored procedures [Cheung et al., 2012]. However, interactive transactions may be trivially included by defining a new operator (e.g., $wait(time)$).
- Transaction type. It specifies the concurrency degree and characteristics of transactions issued by clients according to Kemme [2000]. That is, the amount

of (1) read operations, (2) write operations, (3) conflicting operations ratio, and (4) read/write operation dependencies inside a transaction ratio.

- Replication degree. Amount of nodes in the database that own the same object (e.g., a 100% replication degree means that all servers own all items).
- Hot-spot area size. Amount of objects that are frequently accessed (by read or write operations) by operations [Kemme, 2000].
- Hot-spot area hit frequency. Ratio of operations that access the hot-spot area [Kemme, 2000].

Once all parameters are selected by the user, the configuration script builds the configuration file, mock dataset, and execution schedule as shown in Algorithm 3.1.

The configuration algorithm builds a schedule with as many operations as the transactions per second, number of clients, and number of servers specify. All operations inside each transaction are randomly generated (i.e., buildTrans method in Algorithm 3.1) considering the defined hot-spot area parameters. All objects derived from this schedule are assigned to a set of servers (i.e., updateDataDictionaryFile method in Algorithm 3.1) using a round-robin policy and taking into account the replication degree constraints. Once all transactions are built, all operations in the schedule are shuffled in order to meet the concurrency degree and conflict ratio defined by the transaction pattern parameter. Finally, the script writes the results on two plain-text files to be read by the database simulator: load schedule and data dictionary.

1. Load schedule file. It contains a header with the configuration parameters of the simulator and a tail with the load schedule (i.e., transactions and their operations) to be run by all clients. For instance, a load schedule file modeling 7 fully-replicated servers (replication level 100%), 100 clients issuing 10 long transactions per second, and 5% conflicting operations would look as follows:

```
SIMULATION PARAMETERS:
NumSites=7
NumClients=100
Ntrans=10
TypeTrans=Long
ConflictRatio=5
ReplicationLevel=100
HotSpotAreaSize=30
HotSpotHitFreq=15

SCHEDULE OUTPUT:
ScheduleIn={b3,w3(98)wl, b4,w4(106)wl,b7, w7(198)wl,b6,r6(164)rl,
r4(125)wl,w4(125)wl, b1, r1(46)wl,r6(166)rl, r7(191)rl,w7(181)wl,
r1(49)wl,b8 ,r8(224)wl,b9 , w9(227)wl, b2,r2(64)rl, b0, r0(22)wl,
r0(14)rl,b5,r5(134)rl,r8(219)rl,r6(161)rl,r3(85)rl,w2(53)wl,...}
```

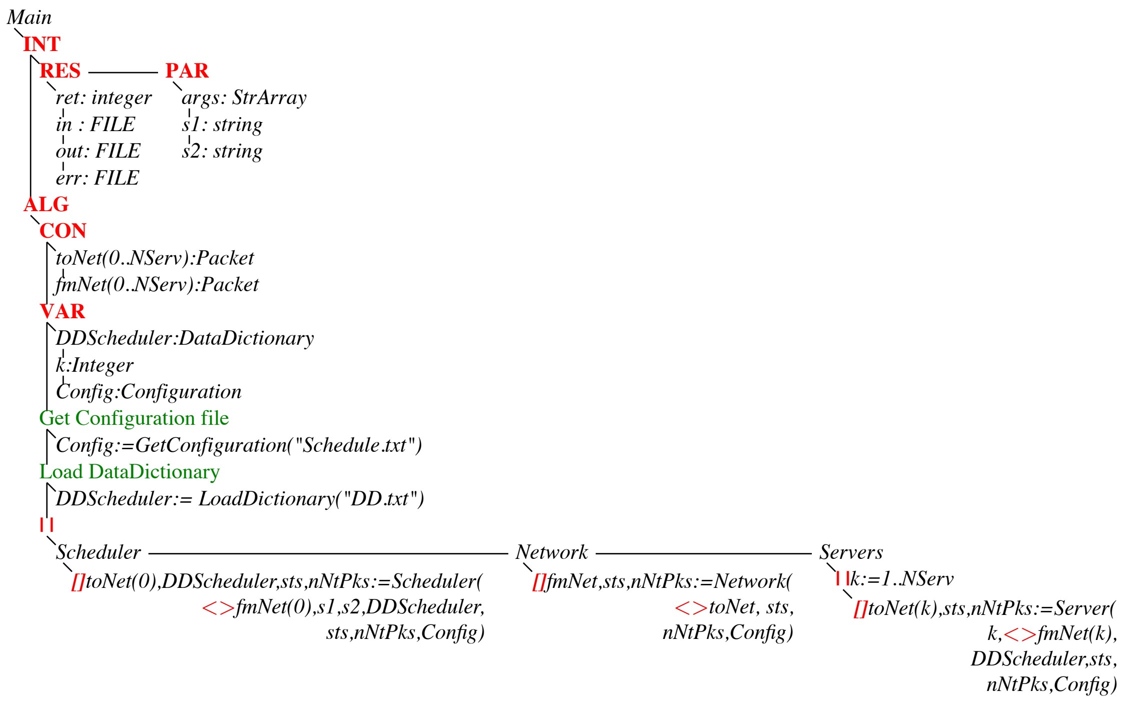

Main
INT
RES
PAR
ret: integer
in : FILE
out: FILE
err: FILE
args: StrArray
s1: string
s2: string
ALG
CON
toNet(0..NServ):Packet
fmNet(0..NServ):Packet
VAR
DDScheduler:DataDictionary
k:Integer
Config:Configuration
Get Configuration file
Config:=GetConfiguration("Schedule.txt")
Load DataDictionary
DDScheduler:= LoadDictionary("DD.txt")
||
Scheduler
[]toNet(0),DDScheduler,sts,nNtPks:=Scheduler(<>fmNet(0),s1,s2,DDScheduler, sts,nNtPks,Config)
Network
[]fmNet,sts,nNtPks:=Network(<>toNet, sts, nNtPks,Config)
Servers
||k:=1..NServ
[]toNet(k),sts,nNtPks:=Server(k,<>fmNet(k), DDScheduler,sts, nNtPks,Config)

Figure 3.2: DFD diagram of the main module of the simulator. The scheduler (clients), servers, and network modules are being executed in parallel.

2. Data dictionary file. It contains the data dictionary file of every server in the database. That is, a correspondence between an object and its location in each server. For instance a data dictionary file modeling 7 servers and fully-replicated objects would look as follows:

```
NUMSITES=10
Object_0:1,2,3,4,5,6#
Object_1:1,2,3,4,5,6#
Object_2:1,2,3,4,5,6#
Object_3:1,2,3,4,5,6#
Object_4:1,2,3,4,5,6#
Object_5:1,2,3,4,5,6#
Object_6:1,2,3,4,5,6#
```

Distributed database simulator. It emulates the behavior of—an arbitrarily large—distributed databases with several clients in a single physical computer according to the parameters provided by the input files described above (load schedule and data dictionary files). As shown in Figure 3.1, the simulator is spread into three independent entities:

1. Network. It is in charge of modeling (1) routing protocols, (2) communication delay effects, (3) GCS, and (4) network congestion. From a logical point of view, the network module can be seen as a set of fully meshed smart hubs that connect all servers among them. Each hub has its own features (e.g., available bandwidth, packet loss probability, packet processing time) in order to model any real network topology.

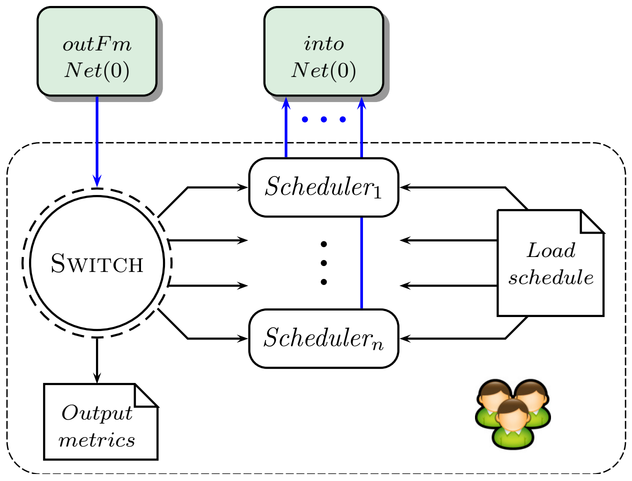


Figure 3.3: Block diagram of the simulator's clients module. Each client is modeled as a transaction scheduler ($\mathtt{Scheduler}_i$) that issues operations to the network and receives the response from a packet switch (Switch).

2. Clients. It is in charge of serially issuing the transactions defined in the load schedule file (i.e., operation O_{i+1} must wait for operation O_i to be issued).
3. Servers. It is in charge of running the selected replication protocol, concurrency control algorithm, and hardware features such as disk storage.

The source code of the simulator is written in PADD/RALE [Babot, 2009; Beltran, 2010], which is a high level abstraction of the American National Standards Institute (ANSI) C language that (1) models real-time distributed thread execution, (2) eases parallel thread programming through intuitive Data Flow Diagrams (DFDs), and (3) ensures reliable throughput and performance measurements. So far, the PADD/RALE framework does not allow real distributed executions in shared nothing environments. Therefore, the simulator has been designed to minimize the amount of system resources while keeping process parallelism and simplicity in a shared memory system.

An overview of the DFD source code that implements the main module of the proposed database simulator is shown in Figure 3.2. After some initializations, we can see that the configuration files are read and the parallel execution [Beltran, 2010] of three entities starts: scheduler (that models clients' behavior), network, and servers.

Results processing. During the simulation, all servers and network hubs count and store their own partial metrics (e.g., transaction response time, transactions' abort ratio, network congestion). When the execution ends, all these partial metrics are aggregated, summarized, and written to a single text file.

The remainder of this section focuses on the simulator internals depicted in Figure 3.1 and details the output results file.

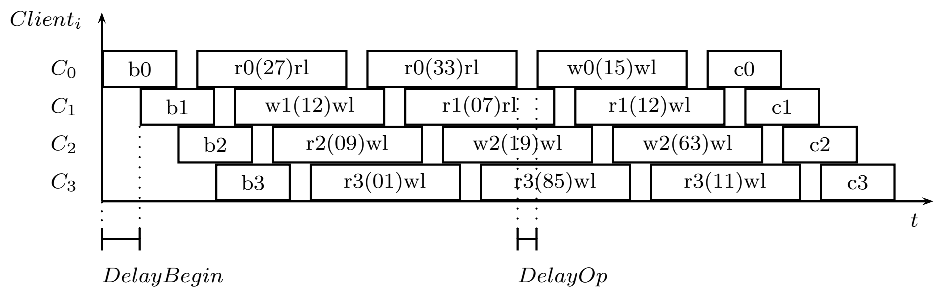


Figure 3.4: Transactions interleaving example. DelayBegin models the time gap between two consecutive transactions issued by the same client. DelayOp models the time gap between two consecutive operations of the same transaction.

3.2.2 *Clients module*

It models the clients' behavior. In order to (1) build a scalable model of the distributed database clients' entity and (2) permit the simulation of an arbitrarily large number of clients, we have split the clients module into two functional blocks shown in Figure 3.3: transaction scheduler and packet switch.

- **Transaction scheduler.** It models the client's functional behavior. Thereby, the simulator runs as many transaction scheduler modules as the number of clients previously configured [Pandis et al., 2011a].
- **Packet switch.** It connects servers and clients in a cost-efficient way. In addition, it also computes some performance metrics—detailed in Section 3.2.6—such as response time, restart ratio, and transactions per second [Kemme, 2000].

Both entities are detailed in what follows.

3.2.2.1 *Transaction scheduler*

The goal of abstracting and modeling the client-side transaction execution is twofold: (1) to provide different degrees of parallelism between transactions and (2) to satisfy the requested concurrency level among clients. We have defined two parameters that tune and quantify the operations'—and thus transactions'—interleaving: DelayBegin and DelayOp. Each parameter is aimed to model the following facts:

- DelayBegin. It is the amount of time before a client executes the first operation of a transaction. To maximize the system stress it has to be set to zero, which means that the client is continuously executing transactions.
- DelayOp. It is the cost in time units of submitting an operation to the database. Large values of this parameter mean that the database is operating with heavy objects such as entire data tables. Hence, DelayOp also models the system granularity, which is the physical characterization of each object inside an operation (e.g., data item, row, column, or table).

The configuration of these values also adjusts the system concurrency. For instance, if DelayBegin is set to

$$\mathsf{DelayBegin} = \mathsf{transactionLength}(\mathsf{DelayOp}) \times \mathsf{number\ of\ clients}, \tag{3.1}$$

Algorithm 3.2 Client functional behavior.

```
Require: Load schedule file is consistent
1: load allOperations from LoadSchedule.txt of this.SchedulerID to ScheduleIn
2: while Operation ≠ NULL do
3:   || get Operation from ScheduleIn
4:      if Operation = Begin then
5:         delay DelayBegin
6:         choose Transaction Coordinator
7:         update Statistics
8:      else
9:         delay DelayOp
10:     end if
11:     send Operation to Transaction Coordinator
12:  || get Operation from Network
13:     update Statistics
14: end while
15: print Simulation Statistics
```

no concurrency will take place. The effects of these two parameters are shown in Figure 3.4. This example illustrates four clients issuing their own transactions. We observe that (1) there is a gap of DelayOp time units between two consecutive operations different from begin or commit, and (2) each client waits DelayBegin time units before issuing another transaction.

Note that we have assumed that each client cannot issue more than one transaction at a time. Hence, to model a client that is able to issue several transactions concurrently, we have to define more clients attached to the same server with a network connection cost of zero [Pandis et al., 2011a].

According to these premises, a client behaves as described in Algorithm 3.2. It first reads the operation from the Load schedule file and then checks whether the operation is the transaction start delimiter. If so, it waits DelayBegin time units and chooses a server to coordinate the transaction (i.e., the server which the client is attached to). Next, it sends the operation to the coordinator. Finally, it waits DelayOp time units before issuing the next operation, and, at the same time (denoted with "||" in Algorithm 3.2), it waits for the response of this operation.

It is worth to note that this approach models the client's behavior—and thus, the distributed database—as a partly-open system [Schroeder et al., 2006]: as shown in Algorithm 3.2 (lines 3–12), new operations are launched to the database no matter whether older ones are processed. In this way, the user can easily adjust an arbitrary load degree (i.e., transactions per second) for the database.

3.2.2.2 *Packet switch*

In order to minimize the number of threads being executed by the simulator [Pandis et al., 2011a], two single network interfaces (also referred to as outFmNet(0) for the input channel and intoNet(0) for the output channel) are provided to the whole clients module. Hence, the packet switch collects every frame coming from any server

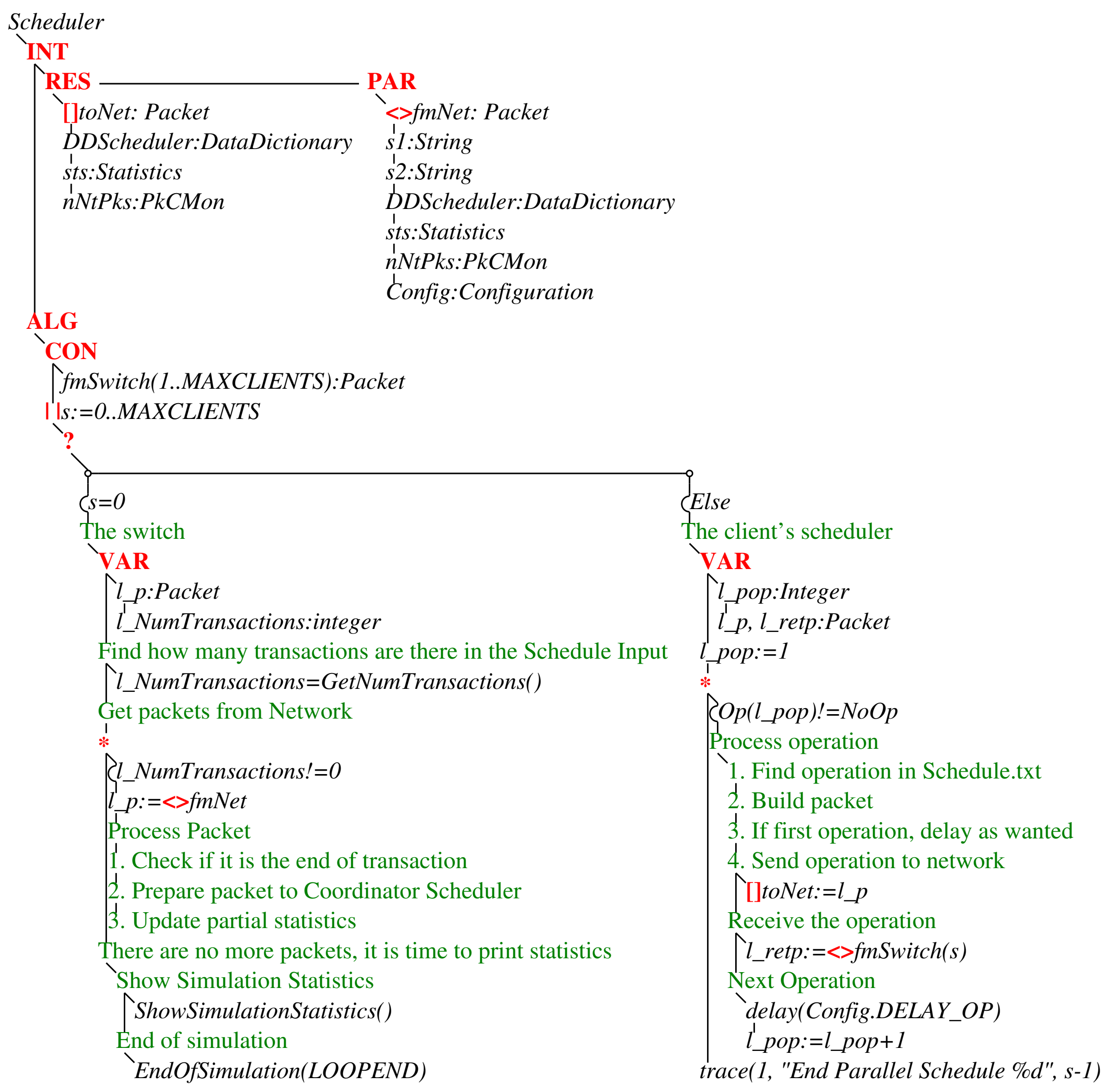

Scheduler
INT
RES
[]toNet: Packet
DDScheduler:DataDictionary
sts:Statistics
nNtPks:PkCMon
PAR
<>fmNet: Packet
s1:String
s2:String
DDScheduler:DataDictionary
sts:Statistics
nNtPks:PkCMon
Config:Configuration
ALG
CON
fmSwitch(1..MAXCLIENTS):Packet
| |s:=0..MAXCLIENTS
?
s=0
The switch
VAR
l_p:Packet
l_NumTransactions:integer
Find how many transactions are there in the Schedule Input
l_NumTransactions=GetNumTransactions()
Get packets from Network
*
l_NumTransactions!=0
l_p:=<>fmNet
Process Packet
1. Check if it is the end of transaction
2. Prepare packet to Coordinator Scheduler
3. Update partial statistics
There are no more packets, it is time to print statistics
Show Simulation Statistics
ShowSimulationStatistics()
End of simulation
EndOfSimulation(LOOPEND)
Else
The client's scheduler
VAR
l_pop:Integer
l_p, l_retp:Packet
l_pop:=1
*
Op(l_pop)!=NoOp
Process operation
1. Find operation in Schedule.txt
2. Build packet
3. If first operation, delay as wanted
4. Send operation to network
[]toNet:=l_p
Receive the operation
l_retp:=<>fmSwitch(s)
Next Operation
delay(Config.DELAY_OP)
l_pop:=l_pop+1
trace(1, "End Parallel Schedule %d", s-1)

Figure 3.5: Simplified DFD of the clients module implementation. There are MAXCLIENTS internal connections to link the packet switch (left branch) and the transaction scheduler (right branch).

and delivers it to its corresponding transaction scheduler. As shown in Figure 3.3, all schedulers deliver their packets to intoNet(0) interface and the packet switch receives the responses from the outFmNet(0) interface.

Recall that the PADD/RALE framework ensures that neither the algorithmic cost, nor the execution cost of the transaction scheduler and the packet switch entities will be reflected on the output metrics computation. Actually, the IDE just considers Delay operations (e.g., lines 5 and 9 in Algorithm 3.2) when computing performance statistics [Babot, 2009; Beltran, 2010].

3.2.3 *Clients module final implementation*

Figure 3.5 depicts the implementation of the clients module in the PADD/RALE framework. After creating a set of costless internal connections (fmSwitch in Figure 3.5) to

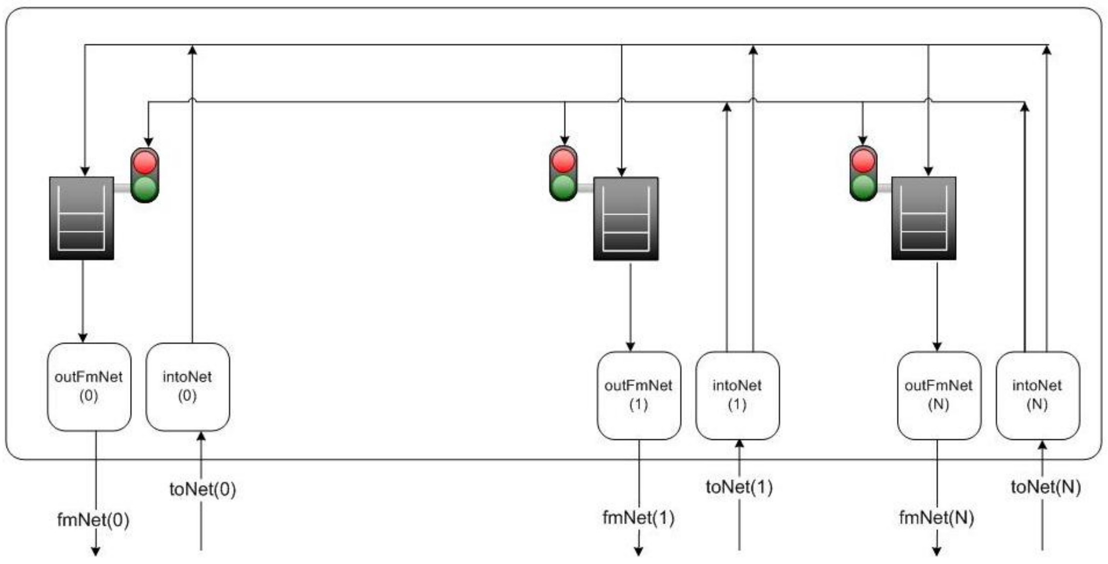


Figure 3.6: Block diagram of the simulator's network module. Each interface is modeled with two entities and a queue managed by a semaphore.

link the packet switch to every client, a set of MAXCLIENTS + 1 parallel processes are created (i.e., the switch plus the MAXCLIENTS schedulers/clients depicted in Figure 3.3). It is worth mentioning that (1) the left branch in Figure 3.5 implements the packet switch, (2) the right branch in Figure 3.5 implements the transactions scheduler, and (3) both branches in Figure 3.5 are executed in parallel.

On the one hand, the packet switch waits for incoming messages from its associated input network interface (fmNet in Figure 3.5) and forwards them to the associated scheduler through the previously defined fmSwitch connection. Recall that this way the number of communication interfaces remains constant—despite the number of clients to be simulated—and, therefore, we are able to minimize both the number of network interfaces and threads [Pandis et al., 2011a].

On the other hand, a set of MAXCLIENTS schedulers are created and run in parallel. Each scheduler sends the packet containing the operation gathered from the *ScheduleIn* file (see Section 3.2)—either begin, read, write, commit, or abort—and its corresponding arguments (e.g., transaction identifier, timestamp, object identifier) to the network through the toNet interface. Once the answer of this message is received through the fmSwitch interface, the client waits DELAY_OP time units (see Section 3.2.2.1) and sends the subsequent operation.

As the simulator is aimed to compute the performance when running transactions rather than keeping track of the exact object values of every datum—as a real database would do—, when a transaction is aborted, the client discards it forever and, thus, it is never reissued. As all transactions are generated upon a common statistic pattern, this assumption does not interfere on the final performance outcome and simplifies the clients' behavior as well. However, all the burden associated to rollback operations—when applicable—on the server's side is considered in case of abort.

Algorithm 3.3 Behavior of the intoNet(i) entity.

```
Require: -
 1: for i := 0 to MAXINTERFACES do
 2:    connect to outFmNet[i].queue
 3: end for
 4: listen connection from toNet(myID)
 5: while 1=1 do
 6:    get packet from toNet(i)
 7:    switch (packet.GCSType)
 8:       case UNICAST:
 9:             outFmNet[packet.destination].push(packet)
10:          break
11:       case MULTICAST:
12:           for all k in packet.destination
13:             outFmNet[k].push(packet)
14:          end for
15:          break
16:       case BROADCAST:
17:           for k := 0 to MAXINTERFACES
18:             outFmNet[k].push(packet)
19:          end for
20:          break
21:       case TO_BROADCAST:
22:          lock outFmNet[∀].queue
23:           for k := 0 to MAXINTERFACES
24:             outFmNet[k].push(packet)
25:          end for
26:          release outFmNet[∀].queue
27:          break
28:       default:
29:          UnknownServiceException
30: end while
```

3.2.4 *Network module*

It links the clients module with the servers module, models the communications network cost, and implements the GCS system primitives. To this concern, each network interface has been modeled with three parallel software entities shown in Figure 3.6: (1) intoNet(i) entity for handling incoming packets at interface i, (2) outFmNet(i) entity for handling outgoing packets at interface i, and (3) a semaphore-managed queue assigned to every outFmNet(i) entity to temporally store network packets.

- **Incoming packets entity (**intoNet(i)**).** Each public method toNet(i) belonging to its associated entity intoNet(i) (see Figure 3.6) is used by client i to send application messages through the network module. The behavior of this entity is detailed in Algorithm 3.3.

Algorithm 3.4 Behavior of the outFmNet(i) entity.

```
Require: SteadyComCost matrix is consistent
1:  $i := 0$
2:  while $i \leqslant$ MAXINTERFACES do
3:      localQueue.listen()
4:      $i := i + 1$
5:  end while
6:  listen connection from fmNet(myID)
7:  while 1=1 do
8:      if ¬LocalQueue.empty then
9:          packet := LocalQueue.pop()
10:         delay ComCost(packet.source, myID, packet.length, packet.GCSservice)
11:         fmNet(i) := packet
12:         deliver fmNet(i)
13:     end if
14: end while
```

First of all, each intoNet(i) entity opens a connection to all queues existing in the network module—the PADD/RALE framework guarantees that connections between processes that are being run in parallel have no impact on the simulated database final performance—(lines 1–3) and waits for its associated client i to join the system. As soon as client i is connected to intoNet(i) entity (line 4), it continuously waits for any incoming messages (also referred to as packets). When a new packet is dropped to intoNet(i), it is immediately pushed into the destination queue(s) according to (1) the packet destination and (2) the selected ϕ GCS primitive (e.g., unicast, multicast, broadcast, total order broadcast). For instance, to send a message using the total order broadcast primitive [Défago et al., 2004] the intoNet(i) module will block all the network queues through the depicted semaphores (see Figure 3.6) and push the message into them. Once the message has been pushed, the semaphores are released (lines 21–27). Note that, although the simulator is now implementing the logical behavior of GCS primitives, it is still not considering their associated overhead [Défago et al., 2004].

- **Outgoing packets entity (**outFmNet(i)**).** Each public method fmNet(i) belonging to its associated entity outFmNet(i) (see Figure 3.6) is used by i to receive application messages through the network module. The behavior of this entity is detailed in Algorithm 3.4.

  Before processing any new messages, outFmNet(i) entity waits for all intoNet(i) entities to be connected to its local queue (lines 1–3 in Algorithm 3.3 and lines 2–5 in Algorithm 3.4). Afterwards, outFmNet(i) entity waits for client i to be connected. Then, it stands for new incoming packets by continuously polling its associated local queue. As soon as it realizes that the queue is not empty, the packet processing procedure starts. This consists of waiting an amount of time (line 10) equal to the cost of moving the packet from its source to its destination in a real network (also referred to as ComCost) taking into account all network specificities discussed such as GCS overhead, packet length, routing and transport protocols. Finally, the $\mathsf{outFmNet}(i)$ module waits for the packet to be read by the process connected to the $\mathsf{fmNet}(i)$ interface (line 12).

We have modeled the communication cost—measured in time units—between two arbitrary network interfaces (m, n) for a packet of ζ bytes long considering the (1) burden derived from stressing a given link, (2) overhead associated to the selected GCS (see Chapter 2 [Hadzilacos and Toueg, 1994]), (3) packet length, and (4) routing and transport protocols load (also referred to as $\mathsf{SteadyComCost}$).

- **Link stress** is derived from the intrinsic behavior of $\mathsf{outFmNet}(i)$ entity detailed in Algorithm 3.4. The more packets are stored in the local queue, the more time it will take for the client/server to receive the desired packet.
- **GCS overhead** stands for the propagation cost of moving a 1-byte-packet from interface m to interface n using the ϕ GCS primitive ($\mathsf{GCSoverhead}(m, n, \phi)$). Hence, it varies according to the selected GCS service. The reader is referred to Défago et al. [2004] and Hadzilacos and Toueg [1994] for detailed costs associated to every GCS primitive.
- **Packet length** includes the network headers and the application layer data.
- **Routing and transport protocols** are modeled by a square matrix—predefined in the configuration file (see Section 3.2.2.1)—that holds the overhead in time units associated to routing and transport protocols for every network link. For instance, Equation 3.2 shows the steady communication cost matrix of a database with i servers:

$$\mathsf{SteadyComCost} = \begin{pmatrix} C_{0,0} & C_{0,1} & \dots & C_{0,i-1} & C_{0,i} \\ C_{1,0} & C_{1,1} & \dots & C_{1,i-1} & C_{1,i} \\ \dots & \dots & \dots & \dots & \dots \\ C_{i-1,0} & C_{i-1,1} & \dots & C_{i-1,i-1} & C_{i-1,i} \\ C_{i,0} & C_{i,1} & \dots & C_{i,i-1} & C_{i,i} \end{pmatrix}, \quad (3.2)$$

  where the $C_{m,n}$ coefficients model the cost of transmitting a packet—without considering neither the GCS facilities nor the link stress—from interface m to interface n. Note that as the clients module is connected to interface 0 of the network module (see Figure 3.3), row $C_{0,k}$ and column $C_{k,0}$ model the cost of sending a message from/to a client respectively.

  As a result, this matrix is able to model any network topology. For instance, costless loopback interfaces can be modeled by setting $C_{m,m} = 0$, asymmetric communication links with α capacity can be modeled by setting $C_{m,n} = \alpha - C_{n,m}$, non-existing links can be modeled by setting $C_{m,n} = -1$, clients locally attached to servers can be modeled by setting $C_{0,n} = C_{n,0} = 0$, or dynamic routing and transport protocols can be modeled through time-variant functions by setting $C_{m,n} = \Gamma(t)$. Nevertheless, as the simulator is typically focused on analyzing the database behavior rather than the network phenomena, it is very common to assume that network behavior is steady over time (neither physical link changes nor network reconfigurations will occur) and, thus, configure the steady communication cost matrix as follows:

$$\mathtt{SteadyComCost} = \lambda * \begin{pmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 1 & 1 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 1 & \dots & 0 & 1 \\ 0 & 1 & \dots & 1 & 0 \end{pmatrix}, \qquad (3.3)$$

that is, a full meshed network with a steady communication cost of λ time units and costless loopback interfaces.

Hence, the communication cost—utilitzed in Algorithm 3.4—of sending a packet of size ζ between network interface m and network interface n using the ϕ GCS primitive is computed as

$$\mathtt{ComCost}(m,n,\zeta,\phi) = \mathtt{SteadyCost}(m,n) + [\zeta * \mathtt{GCSoverhead}(m,n,\phi)]. \qquad (3.4)$$

Finally, let us check whether this network model satisfies the dynamic group point-to-point communication formal properties stated by Schiper [2006], which will contribute to check the correctness of the protocols implemented on the simulated database.

- **Channel validity.** If a process $\mathtt{outFmNet}(i)$ receives a message m, then m has to be sent by some process $\mathtt{intoNet}(k), k = \{0, 1, ..., i-1, i, i+1, ..., n\}$. This is ensured by the fully-deterministic behavior of the PADD/RALE framework.

- **Channel nonduplication.** A process $\mathtt{outFmNet}(i)$ has to receive message m at most once. This is guaranteed by the $\mathtt{intoNet}(k)$ module, which correctly places m to its corresponding queues.

- **Channel termination.** If a process $\mathtt{toNet}(i)$ sends a message m to another interface $\mathtt{fmNet}(k)$, and $\mathtt{intoNet}(i)$ and $\mathtt{outFmNet}(k)$ are both correct, then $\mathtt{fmNet}(k)$ eventually delivers m. This is confirmed since (1) $\mathtt{intoNet}(i)$ is always filling $\mathtt{outFmNet}(k)$'s queues, and (2) $\mathtt{outFmNet}(k)$ is continuously processing its queue.

- **FIFO order.** If some process $\mathtt{toNet}(i)$ sends a message $m1$ before sending $m2$ to interface k and $\mathtt{fmNet}(k)$ delivers $m2$, then $\mathtt{outFmNet}(k)$ must have already delivered $m1$ previously. This is trivially provided by the definition of the $\mathtt{outFmNet}(k)$'s queue.

Therefore, we can assure that all communication formal properties stated by Schiper [2006] are satisfied. The servers module is described in what follows.

3.2.5 *Servers module*

It models all duties concerning data storage and retrieval in distributed databases. While most of the modules of this simulator are pretty standard and do not require software modifications, practitioners are encouraged to tweak the servers module when implementing new protocols. To this concern, we have followed a modular approach

Algorithm 3.5 Basic server's functional behavior.

```
Require: Data Dictionary file is consistent
 1: load DataDictionary
 2: while true do
 3:   get Operation from Network
 4:   if $Operation_{ij} = N_i$ then
 5:     execute (Termination Protocol,CCAPolicy)
 6:   else
 7:     IsLocal := CheckOperation($Operation_{ij}$,DataDictionary)
 8:     execute ($Operation_{ij}$,IsLocal,CCAPolicy)
 9:     delay DelayIO
10:   end if
11:   send $Operation_{ij}$.Status to Scheduler
12: end while
```

similar to the one proposed by Bernstein et al. [1987] depicted in Figure 2.1 and split the servers module into three entities: persistent data repository, data dictionary, and concurrency control and replication management.

- **Persistent data repository.** It simulates the cost associated to store and retrieve any datum to/from memory (e.g., main memory, hard disks). The final outcome of this module is an integer value that represents the cost of applying the requested operation (i.e., read or write). Other effects—out of the scope of this dissertation—such as file system type, storage technology, or disk cache behavior can be modeled as well. For instance, the behavior of an in-memory database [Jones et al., 2010] could be modeled by assigning low values to the cost of applying updates. In addition, data granularity (i.e., data registers, or data rows, or data columns, or data tables) can be modeled by adjusting a constant value belonging to this entity.
- **Data dictionary.** It holds the metadata [Ghemawat et al., 2003] associated to the location(s) of any object stored in the simulated database. This module is initialized with the values collected from the data dictionary file (see Section 3.2.1) and updated accordingly during the simulation. Note that this entity is only used in those scenarios that require partial replication [Bernabé-Gisbert et al., 2008] (i.e., there is no uniform distribution of data objects over all servers).
- **Concurrency control and replication management module.** It implements the message exchange protocols associated to the selected concurrency control algorithms and replication protocols.

The behavior of the server module is detailed in Algorithm 3.5. First, all servers load the data dictionary file in order to know the location of all objects in the database. Afterwards, each server waits for any incoming packets (line 3 in Algorithm 3.5) from its corresponding input network interface fmNet(k) (see Figure 3.6). When a packet is received, the server checks whether it corresponds to a termination action (also referred to as N_i for transaction i).

If the operation contained inside the received packet signals the end of a transaction (i.e., commit or abort), the server initiates the transaction termination procedure (line

5 in Algorithm 3.5)—defined by the concurrency control algorithm and replication protocol—to synchronize all replicas and avoid consistency issues.

On the contrary, if the j-th operation is a $\mathtt{read}$ or $\mathtt{write}$ on $\mathtt{object}_o$ belonging to transaction i ($\mathtt{Operation}_{ij}$) (lines 7–9 in Algorithm 3.5), the server checks whether it can be executed locally (i.e., $\mathtt{object}_o$ is contained inside the local data dictionary) and executes the operations according to the replication strategy. If the transactions cannot be executed locally or other sites need to update $\mathtt{object}_o$, the operation is deferred through the output network interface $\mathtt{intoNet}(k)$ (see Figure 3.6) to a site that contains the requested object.

The execution of any operation—coming from a local or a remote request—on the server is modeled as an access to the persistent storage module. This module delays *DelayIO* time units (line 9 in Algorithm 3.5) the execution of all subsequent operations on the server according to the selected persistent storage policy (e.g., main memory or secondary storage). Note that in some particular situations (e.g., optimistic concurrency control), the termination protocol may include some additional accesses to the persistent storage module as well; to reduce the number of disk accesses and avoid rollback actions, servers may hold the operations of every transaction at main memory until the termination operation, and move them to disk in case of commit.

Finally, the server sends back the resulting execution status (line 11 in Algorithm 3.5) to the client (i.e., scheduler). At this point, some performance metrics detailed in the following subsection are updated.

3.2.6 *Performance metrics*

Selecting the appropriate feature or parameter of the distributed database simulator to obtain significant outcomes is sometimes difficult. Hereafter, a discussion about where metrics should be collected and their meaning is provided. We distinguish two major analysis sources: servers and network [Curino et al., 2012]. While parameters measured at the servers side measure the effectiveness of the replication and concurrency protocols, network metrics are mostly used to know whether the communication link is acting as a bottleneck or not. Hence, the simulator includes two measuring modules: the network sensing entity and the servers sensing entity.

Network sensing entity. This entity—included inside the network module shown in Figure 3.1—is aimed to measure the network congestion of the distributed database.

Servers sensing entity. This entity—included inside the clients module shown in Figure 3.1—is aimed to measure the performance of all servers in the distributed database.

The rationale of placing this entity inside the clients entity rather than inside the servers entity is three-fold: (1) to isolate the servers development, which is continuously updated, from the clients entity, which is rarely modified, (2) to concentrate all the measures into a single entity, and (3) to emulate real-world experimentation where databases are often inaccessible and, thus, metrics are collected on the clients side [Curino et al., 2012].

Specifically, the network sensing entity computes the following performance metrics:

Number of packets in the network (NPN)**.** It is the main indicator of network usage and contention. This metric quantifies the amount of packets that are traveling on the network per unit of time. Accordingly, this value is computed as the sum of packets waiting in every network queue (see Figure 3.6) every second.

An exceedingly high value of NPN means that the network cannot cope with the communications traffic required for the system (i.e., the implemented replication and concurrency protocols are generating so many messages that the network is unable to handle them). An exaggerated low value of NPN may point that the network is oversized or that there is not enough traffic to saturate it.

Packet delay (PD)**.** This metric quantifies the amount of time in seconds that a packet spends on crossing the network module. To measure it, a timestamp (labeled as $TS(MsgIn_i)$) is added every time a packet is pushed into the queue managed by the outFmNet entity associated to network interface i (see Figure 3.6) and another timestamp (labeled as $TS(MsgOut_i)$) is assigned as soon as the message leaves the queue. Then PD is computed as shown in Equation 3.5:

$$PD[seconds] = TS(MsgOut_i) - TS(MsgIn_i). \tag{3.5}$$

Actually, PD and NPN are closely related in the sense that both parameters will increase and decrease simultaneously. Nonetheless, PD includes some extra information when analyzing the sources for network congestion. For instance, looking at PD it is possible to assert whether the network is spending too much time on processing every packet (i.e., the communication cost in Equation 3.4 is too high) or it is saturated due to the overwhelming number of delivered packets.

In addition, it is possible to measure some other metrics such as queue length (i.e., maximum number of packets stored on a network queue), interface usage (i.e., number of packets that cross a given interface), or communication patterns (i.e., number of packets from network interface k that go to interface j) to obtain a finer analysis of the network performance. However, conducted experiments so far show that taking the average values of NPN and PD is enough to extract significative results.

Similarly, the servers sensing module computes the following performance metrics, which are similar to the ones proposed in Kemme [2000] and Curino et al. [2012]:

Throughput (THR)**.** This metric quantifies the amount of transactions that the simulated database has been able to execute every second (also referred to as transactions per second (tps) for short). It is computed as the sum of commits and aborts received by all clients during every second as shown in Equation 3.6:

$$THR[tps] = \frac{1}{Simulation\ time} * \sum_{i=1}^{MAXCLIENTS} Commits_i + Aborts_i. \tag{3.6}$$

Ideally, this measure should be equal to the number of transactions per second that clients issue. However, there exists a certain threshold where the system is unable to process all transactions and thus THR plunges [Gray et al., 1996; Serrano et al., 2007].

Note that THR takes into account both, committed and aborted transactions (i.e., the rough amount of work carried by the distributed database), which hides the (1) efficiency of the concurrency control algorithm—a dummy protocol that aborts all transactions may obtain great THR values—, and (2) competency of the replication protocol. Therefore, the following metric is defined.

Restart Ratio (RR)**.** This metric computes the ratio of aborted transactions versus the total number of issued transactions. It is computed as follows:

$$RR[\%] = \sum_{i=1}^{\text{MAXCLIENTS}} \frac{\mathsf{Aborts}_i}{\mathsf{Commits} + \mathsf{Aborts}} * 100. \qquad (3.7)$$

Assuming that all transactions are aimed to commit, a high RR value means that there is a considerable number of transactions that are being aborted (due to a high concurrency degree) by the distributed database. Note that if aborted transactions were allowed to be issued again, the RR value would have a close relationship with the transactions' reissue policy [Kemme, 2000].

Transaction Time (TT)**.** This metric, also referred to as transactions response time, computes the average duration—in time units—of every transaction. It is computed since its begin operation is sent to the network until its $\mathsf{commit/abort}$ operation is received by the client:

$$TT[\text{seconds}] = \frac{1}{m} \sum_{i=1}^{m} TS(N_i) - TS(b_i). \qquad (3.8)$$

As transactions are considered to as stored procedures [Cheung et al., 2012] this metric enables the measurement of the overhead added by each replication and concurrency protocol. Therefore, a low value of TT means that the protocol is fast at executing transactions.

Note that this parameter has to be considered next to the RR; a trivial concurrency protocol that aborted all transactions without further verifications (i.e., returning an abort as soon as the delegate replica receives the commit operation) would get the lowest TT and the poorest RR as well.

This section concludes the simulator description. In what follows, a set of replication and concurrency protocols implemented over this simulator and their performance are presented.

3.3 CONCURRENCY CONTROL AND REPLICATION ALGORITHMS IMPLEMENTATION

In order to show the simulator's performance, we have implemented the following concurrency control and replication protocols: Non-Distributed Concurrency Control (NDCC), BOCC, Basic Replication Protocol (BRP), which are based on the principles detailed in Chapter 2. In this regard, the description of each protocol implementation is based on a brief pseudocode and a message diagram, as done in [van Renesse and Guerraoui, 2010]. Finally, the implementation guidelines to deploy other replication schemes [Kemme, 2000; Wiesmann and Schiper, 2005] with different isolation levels are provided.

3.3.1 *Non-Distributed Concurrency Control algorithm implementation*

NDCC is a mock concurrency control algorithm that ignores any consistency issues: as soon as operations are issued to the system they are executed without further verifications. Hence, it is used to (1) verify that the simulator is working properly, (2) obtain ideal performance values, and (3) measure the overhead added by other modules (i.e., clients and network). Its behavior and message exchange protocols when processing a begin, read, write/update, and termination operation is detailed in what follows.

TRANSACTION BEGIN OPERATION: When the client issues a begin operation, it—randomly—selects an available server (also referred to as delegate replica) and notifies it that it will be in charge of coordinating the remainder of the transaction. This is achieved by sending a lightweight message to the delegate replica that includes the transaction identifier i as shown in Figure 3.7.

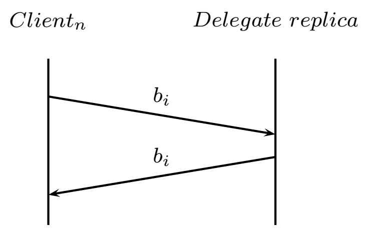


Figure 3.7: Transaction begin operation for Non-Distributed Concurrency Control.

TRANSACTION READ OPERATION: When a read operation is issued to the delegate replica it first checks whether the requested object is stored in its local database. If object x_k is stored in the delegate replica's local database, it executes the read operation as detailed in Algorithm 3.5. On the contrary, if the object is stored at another server, the delegate replica looks for an eligible replica ($Owner_j$) at its local data dictionary file and defers it the read operation ($dr_i(x_k)$). Once the read is complete, either local or remote, the client is notified as shown in Figure 3.8.

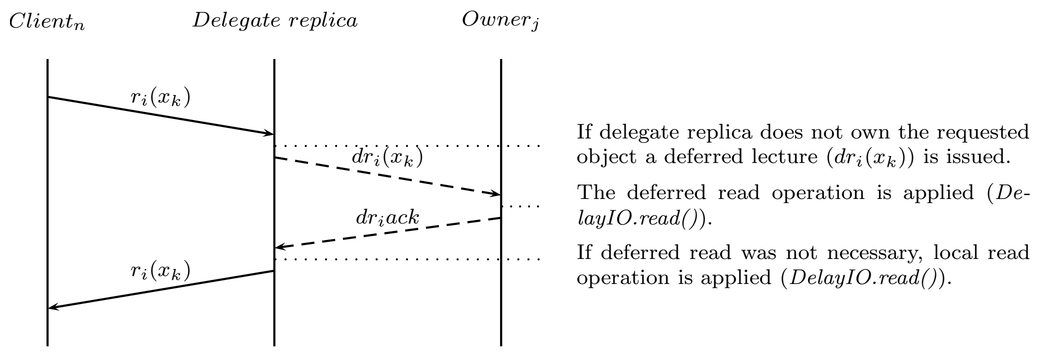

$Client_n$
Delegate replica
$Owner_j$
$r_i(x_k)$
$dr_i(x_k)$
dr_iack
$r_i(x_k)$
If delegate replica does not own the requested object a deferred lecture ($dr_i(x_k)$) is issued.
The deferred read operation is applied (*DelayIO.read()*).
If deferred read was not necessary, local read operation is applied (*DelayIO.read()*).

Figure 3.8: Transaction read operation for Non-Distributed Concurrency Control.

TRANSACTION WRITE/UPDATE OPERATION: When a write operation is issued to the delegate replica it first looks for all servers that own object x_k. Then, the delegate replica defers them the update operation—including itself—to keep all replicas synchronized. Later, each server executes the update operation as detailed in Algorithm 3.5. Once all replicas notify the delegate replica that they have finished the write operation, the reply is forwarded to the client as shown in Figure 3.9.

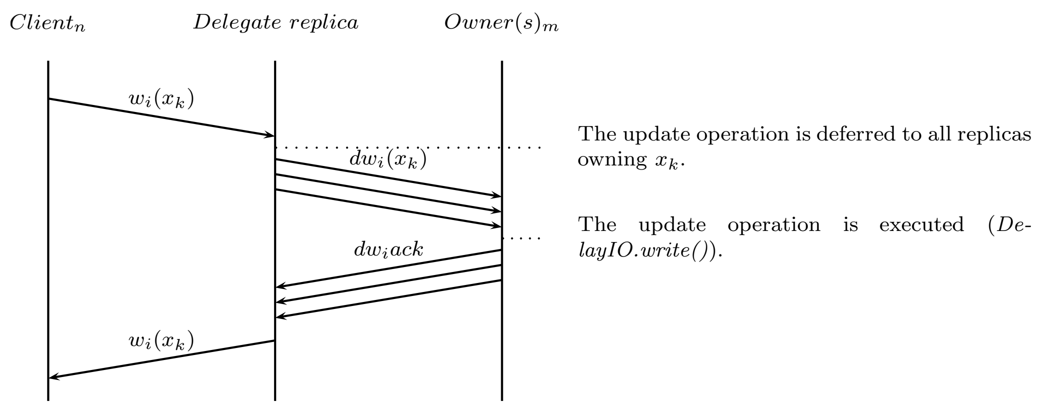

$Client_n$
Delegate replica
$Owner(s)_m$
$w_i(x_k)$
$dw_i(x_k)$
$dw_i ack$
$w_i(x_k)$
The update operation is deferred to all replicas owning x_k.
The update operation is executed (DelayIO.write()).

Figure 3.9: Transaction write/update operation for Non-Distributed Concurrency Control.

TRANSACTION TERMINATION OPERATION: When a termination operation N_i is issued to the delegate replica it first looks for all servers that have participated [Bernstein et al., 1987] in the transaction. Then, it broadcasts them—including itself—a message to inform that transaction i finished (note that at this point, if concurrency control were enabled, each participant should release all locks related to transaction i objects). Later, all participants acknowledge to be aware of the transaction i termination to the delegate replica. Finally, the acknowledge is forwarded to the client as shown in Figure 3.10.

When running the NDCC algorithm, all transactions are forced to commit (i.e., no aborts are permitted) in order to obtain an upper threshold of the system performance. Actually, if aborts were considered, update operations of aborted transactions should be rollbacked, and, thus, the performance would be reduced considerably (i.e., a rollback operation has the same cost than an update operation).

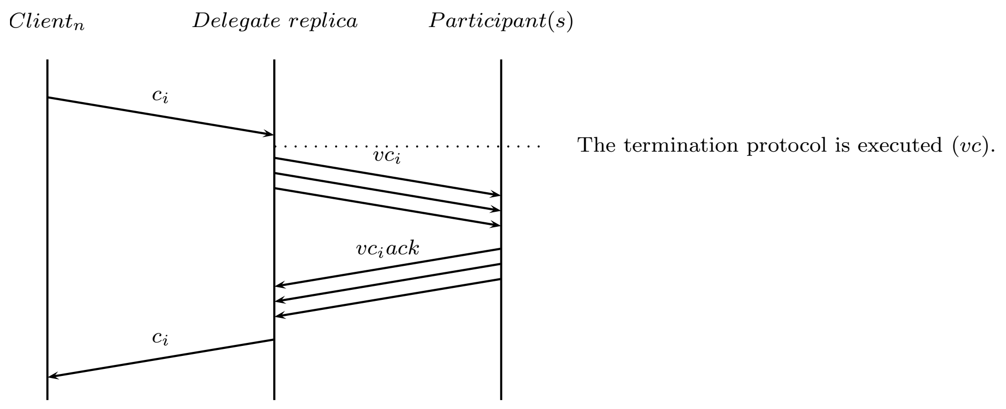


Figure 3.10: Transaction termination operation for Non-Distributed Concurrency Control.

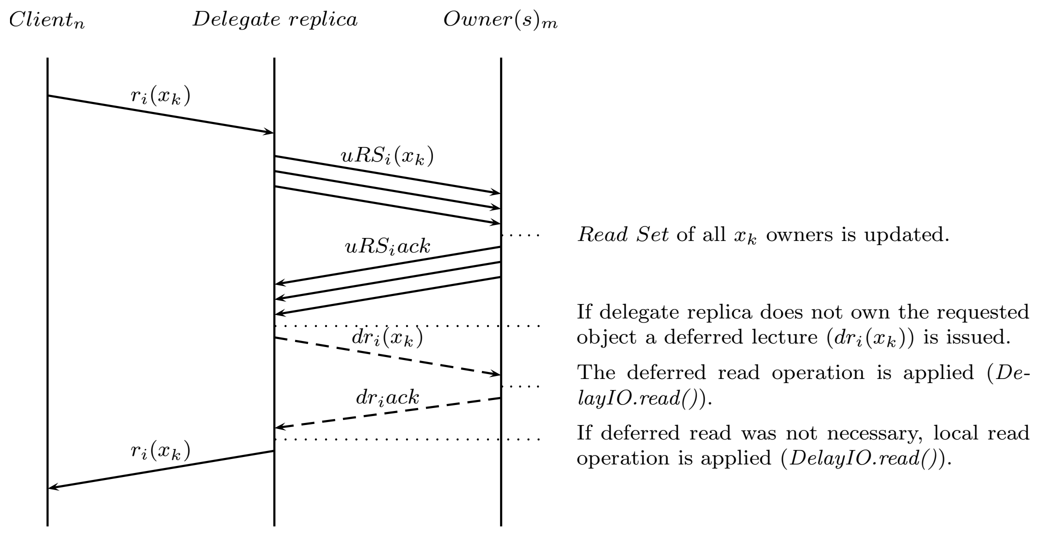
3.3.2 *Basic Optimistic Concurrency Control algorithm implementation*

BOCC is an optimistic concurrency control (see Chapter 2) for partial replication schemes based on the O2PL [Özsu and Valduriez, 1999] protocol proposed in Kemme [2000]. It is used (1) as a first approach to concurrency control in partially replicated databases, (2) to illustrate the behavior of optimistic concurrency control, and (3) to highlight the main challenges of partial replication [Kemme, 2000; Bernabé-Gisbert et al., 2008]. For the sake of this dissertation, it is designed to provide the highest isolation degree (i.e., serializable) and, and, thus, generate 1SR histories. Its behavior and message exchange protocols when processing a begin, read, write/update, and termination operation are detailed in what follows.

TRANSACTION BEGIN OPERATION: When the client issues a begin operation, the system behaves in the same way as defined for the NDCC algorithm shown in Figure 3.7.

TRANSACTION READ OPERATION: When a read operation is issued to the delegate replica it first looks for all servers that own object x_k. Then, the delegate replica sends a message to inform all object x_k owners that they have to update the RS associated transaction i ($uRS_i(x_k)$) (also referred to as read lock). Afterwards, if object x_k is stored in the delegate replica's local database, it executes the read operation as detailed in Algorithm 3.5. On the contrary, if the object is stored at another server, the delegate replica looks for an eligible replica ($Owner_j$) at its local data dictionary file and defers it the read operation ($dr_i(x_k)$). Once the read is complete, either local or remote, the client is notified as shown in Figure 3.11.

Note that the extra round of messages ($uRS_i(x_k)$) added prior executing the read operation (1) enables all replicas to lock object x_k and (2) lets the system to be fault tolerant. Specifically, when an $Owner_m$ site fails, the delegate replica may decide—according to the selected replica failure model (e.g., byzantine, crash/stop) [Cristian, 1991]—to exclude $Owner_m$ from the transaction and inform all other replicas in the system that $Owner_m$ is faulty. If no faulty servers were assumed, the result of the read operation could be include in the $uRS_i ack$ message as done in [DeCandia et al., 2007].



$Client_n$
Delegate replica
$Owner(s)_m$
$r_i(x_k)$
$uRS_i(x_k)$
Read Set of all x_k owners is updated.
$uRS_i ack$
If delegate replica does not own the requested object a deferred lecture ($dr_i(x_k)$) is issued.
$dr_i(x_k)$
The deferred read operation is applied (DelayIO.read()).
$dr_i ack$
If deferred read was not necessary, local read operation is applied (DelayIO.read()).
$r_i(x_k)$

Figure 3.11: Transaction read operation for Basic Optimistic Concurrency Control.

TRANSACTION WRITE/UPDATE OPERATION: When a *write* operation is issued to the delegate replica it first looks for all servers that own object x_k. Then, the delegate replica sends a message to inform all object x_k owners that they have to update the WS associated transaction i ($uWS_i(x_k)$) (also referred to as write lock). Then, it defers all $Owner_m$ replicas the update operation—including itself—to keep all replicas synchronized.

Later, each server executes the update operation as detailed in Algorithm 3.5 but stores the result in main memory (*DelayIO.writeMainMemory()*)—instead of storing it to disk (*DelayIO.write()*). Note that *DelayIO.writeMainMemory()* $\ll$ *DelayIO.write()* (see Section 3.2.5). Hence, subsequent accesses of transaction i towards x_k will be forwarded to main memory until transaction i termination. This strategy (1) increases the performance of the rollback operation in case of transaction abort (no disk accesses will be required), and (2) still keeps the system on the serializable isolation level [Berenson et al., 2007]. Once all replicas notify the delegate replica that they finished the *write* operation, the reply is forwarded to the client as shown in Figure 3.12.

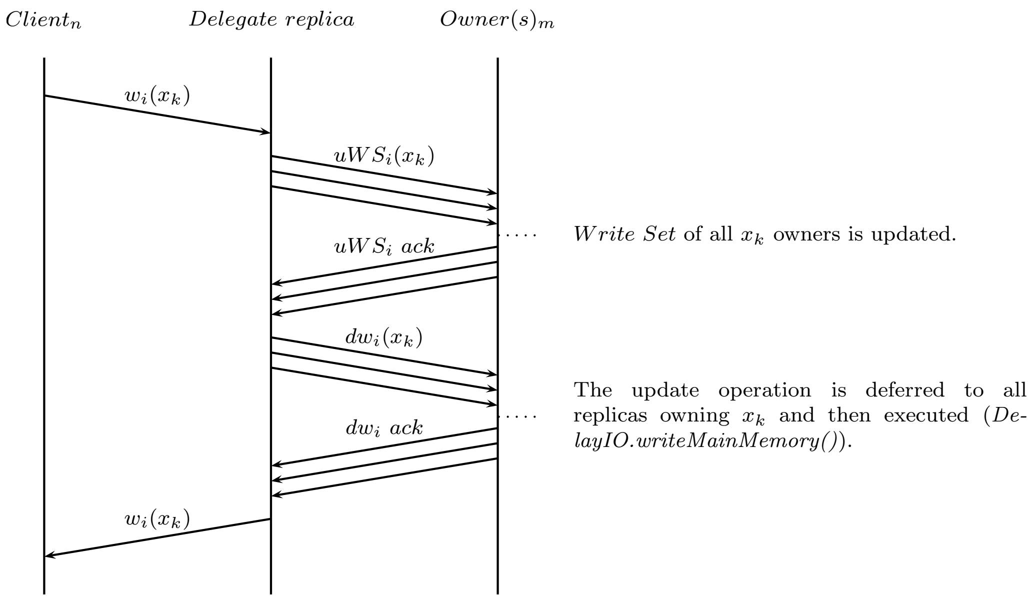

$Client_n$
Delegate replica
$Owner(s)_m$
$w_i(x_k)$
$uWS_i(x_k)$
uWS_i ack
Write Set of all x_k owners is updated.
$dw_i(x_k)$
dw_i ack
The update operation is deferred to all replicas owning x_k and then executed (DelayIO.writeMainMemory()).
$w_i(x_k)$

Figure 3.12: Transaction write/update operation for Basic Optimistic Concurrency Control.

TRANSACTION TERMINATION OPERATION: When a termination operation N_i is issued to the delegate replica it first looks for all servers that participated [Bernstein et al., 1987] in the transaction. If N_i is an *abort* operation, the delegate replica asks all participants to release the locks corresponding to transaction i. Afterwards the client is notified that transaction i has successfully terminated.

On the contrary, if N_i is a commit operation, the delegate replica broadcasts a message to all participants—including itself—for advertising them that transaction i is finished. At this point, each participant checks for possible conflicts between transaction i and all other active (i.e., not terminated) local transactions. Taking into account the 1SR correctness criterion [Bernstein et al., 1987], no intersections are allowed between the (1) WS of transaction i and WS of any other active transaction j, (2) WS of transaction i

and RS of any other active transaction j, and (3) RS of transaction i and WS of any other active transaction j. That is:

$$T_i \text{ consistent} = \begin{cases} \text{Yes} & \text{iff} \quad ((WS_j \cap (WS_i \cap RS_i)) \cup (WS_i \cap WS_j) \neq \emptyset \quad \forall j \neq i \\ \text{No} & \text{otherwise} \end{cases} . \quad (3.9)$$

If no local conflicts are found, the participant will send a vote commit acknowledge ($vc_i ack$) message to the delegate replica. Analogously, if local conflicts are found a vote commit not acknowledge ($vc_i nack$) will be sent. Then, if the delegate replica receives a $vc_i nack$ message it will order all participants to abort transaction i, and release all locks corresponding to transaction i (dnc_i) and notify the client that the commit operation has ended with an abort (a_i) as shown in Figure 3.13. On the contrary, if all participants acknowledge to not have found local conflicts, the delegate replica will order them to make execute the commit operation (dc_i) and make all changes of transaction i durable (i.e., flush the WS to disk). Finally, the client is notified that the commit operation (c_i) has been successfully applied.

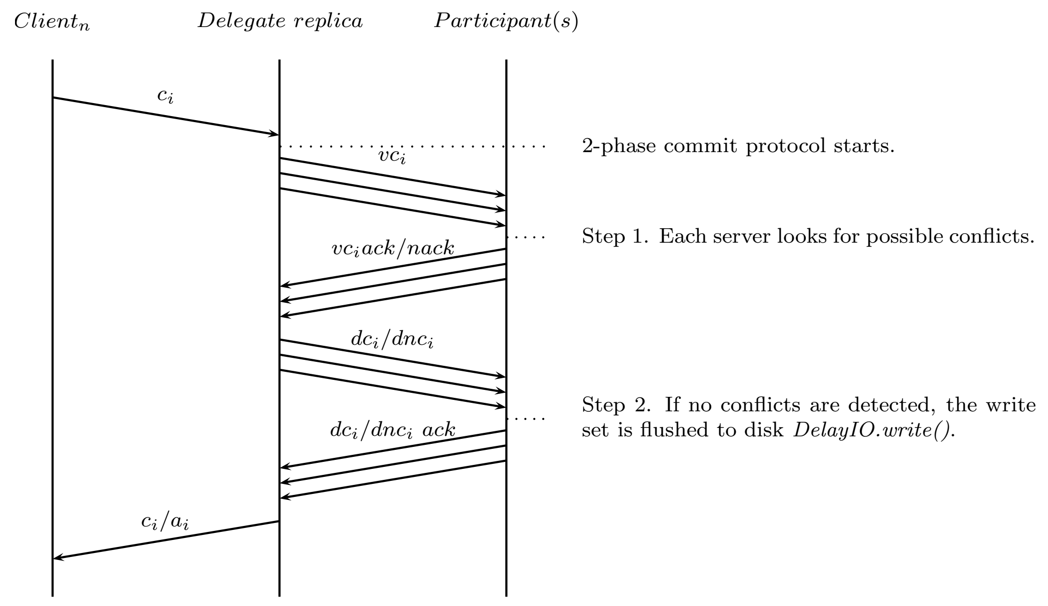

$Client_n$
Delegate replica
Participant(s)
c_i
2-phase commit protocol starts.
vc_i
Step 1. Each server looks for possible conflicts.
$vc_i ack/nack$
dc_i/dnc_i
Step 2. If no conflicts are detected, the write set is flushed to disk DelayIO.write().
dc_i/dnc_i ack
c_i/a_i

Figure 3.13: Transaction termination operation for Basic Optimistic Concurrency Control.

3.3.3 *Basic Replication Protocol implementation*

Locks raised by the BOCC algorithm entail a twofold problem. On the one hand, they prevent several concurrent transactions to successfully commit—also referred to as *last commit wins*: if a transaction locks all objects and never ends, no more transactions will be allowed to progress on the system—, which reduces the database throughput (see Section 3.2.6). On the other hand, transactions that are going to abort—because they are attempting to access locked objects—keep on generating network messages, which adds an extra overhead on the network module.

Therefore we have proposed an evolved version of the BRP detailed in Armendáriz-Iñigo [2006], which includes partial replication support. This algorithm alleviates the

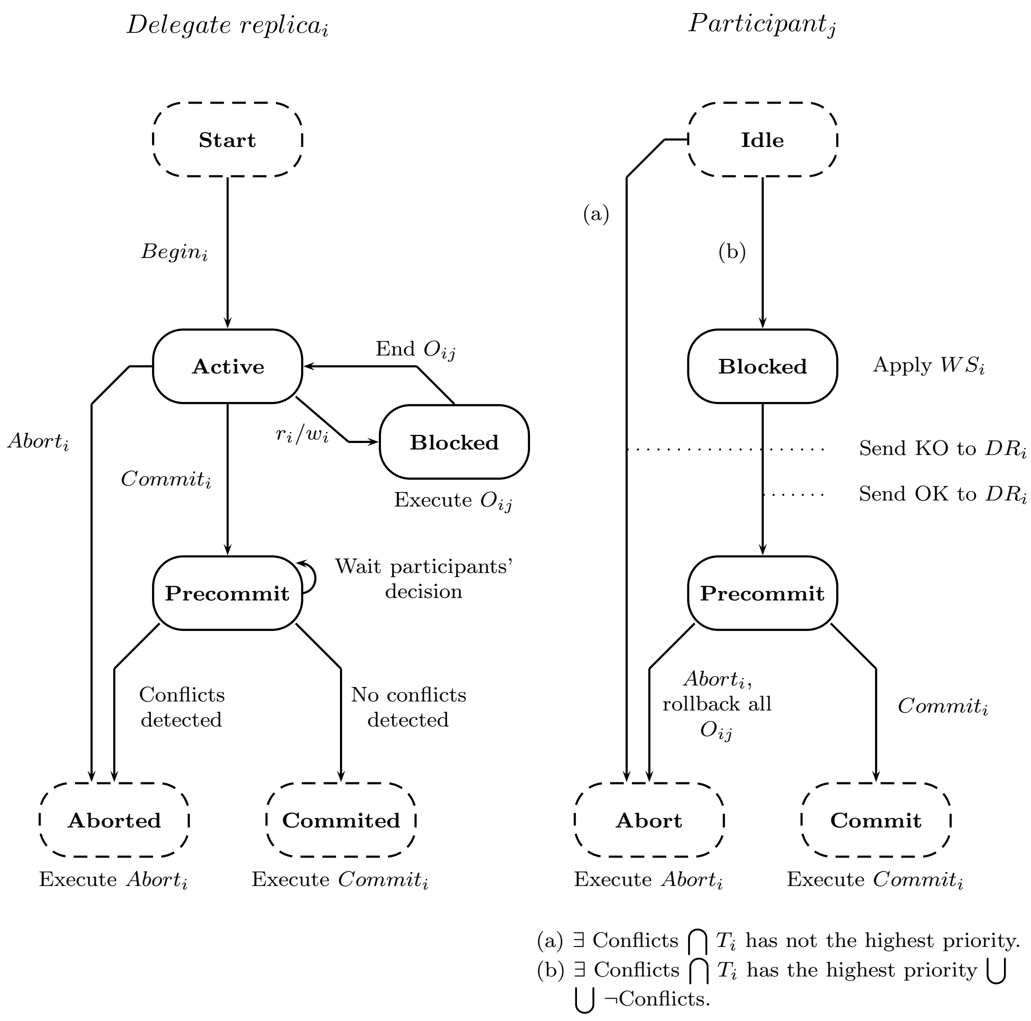


Figure 3.14: Finite state machine diagram of the Basic Replication Protocol.

aforesaid BOCC issues by (1) implementing a priority based commit (e.g., *first commit wins*) and (2) aborting all conflictive transactions as soon as possible. Its behavior when processing a begin, read, write/update, and termination operation is detailed in what follows.

TRANSACTION BEGIN OPERATION: When the client issues a begin operation, the system behaves in the same way as defined for the NDCC and BOCC algorithms shown in Figure 3.7. Optionally, according to the selected commit policy, the delegate replica may decide to store some extra information such as the transaction i start timestamp. Also, this operation prepares the client to receive an unexpected abort (also referred to as unilateral abort) from the selected delegate replica.

TRANSACTION READ AND WRITE/UPDATE OPERATIONS: When a read or write operation is issued to the delegate replica it behaves in the same way as defined for the BOCC algorithm shown in Figure 3.11 and Figure 3.12 respectively. That is, the RS and WS of all participants are updated and write operations are executed in main memory.

VOLUNTARY TRANSACTION TERMINATION OPERATION: As shown in the finite state machine diagram depicted in the left side of Figure 3.14, when the client issues a termination operation N_i to the delegate replica (also referred to as voluntary transaction termination), the delegate replica behaves similarly to the BOCC algorithm detailed in Figure 3.13: the WS is sent to all participants in order to let them check for possible conflicts.

The BRP includes a priority policy on the conflict detection process. Besides the conditions stated in Equation 3.9 that guarantee 1SR correctness criterion, a priority hierarchy is defined among transactions. In order to obtain an algorithm comparable to the the BOCC, we have chosen an aging policy, which gives the highest priority to the transaction with the lowest starting timestamp.

As shown in the right side of Figure 3.14, when a participant is checking for conflicts with transaction i, and i has the highest priority—according to the selected commit policy—among all conflictive transactions, or no conflictive transactions exist, the participant will jump to the $\mathtt{Blocked}$ state. Otherwise, transaction i will be aborted and, thus, the delegate replica will (1) free the main memory space used to store the WS of transaction i, (2) notify the delegate replica DR_i that conflicts have been found and could not be resolved (i.e., $vc_i nack$ in Figure 3.13), and (3) jump to the $\mathtt{Abort}$ state.

When the participant enters the $\mathtt{Blocked}$ state it speculatively flushes the WS of transactioni to disk (*DelayIO.write()*) and tells the delegate replica that no conflicts have been found (i.e., it is assuming that conflicts with less priority transactions will be solved later). Then, the participant will change its state to $\mathtt{Precommit}$ as shown in the right side of Figure 3.14. At this state, the participant waits for the delegate replica to notify whether other participants found any conflicts (i.e., dc_i/dnc_i in Figure 3.13).

If any participant reports that it is unable to resolve transaction i conflicts (i.e., transaction i has not the highest priority), the delegate replica will send a dnc_i message to all participants. Then, they (1) will rollback the WS operations flushed at the $\mathtt{Blocked}$ state and (2) abort transaction i.

On the contrary, if all participants report that they are able to solve conflicts (i.e., transaction i has the highest priority) the delegate replica will send a dc_i message to all participants. Then, they (1) will solve conflicts by sending an unilateral abort message to the associated delegate replica of each conflicting operation, and (2) acknowledge the delegate replica to commit the transaction.

UNILATERAL TRANSACTION TERMINATION OPERATION: On the one hand, early aborting transactions (i.e., terminating them before the client issues the N_i operation) leads to some slight changes in the client's behavior detailed in Algortihm 3.2. Specifically, the client has to be ready to receive unexpected messages from the delegate replica notifying that a transaction has aborted and, thus, none of its operations will be accepted in the future.

On the other hand, when a delegate replica receives an unilateral abort, it has to (1) purge all operations (i.e., WS and RS) and local variables related to the requested transaction, and (2) inform the client that the transaction has been terminated by the database.

3.3.4 *Other replication techniques*

Other replication strategies [Kemme, 2000; Wiesmann and Schiper, 2005]—or modifying existing ones—can be easily implemented upon the aforesaid modules. For instance, we have assumed a 1SR update-everywhere eager replication strategy (see Section 2.3) so far. In what follows, the implementation guidelines for their analogous protocols 1CSI primary-backup and 1SR lazy replication protocols are detailed.

1CSI PRIMARY-BACKUP REPLICATION PROTOCOL IMPLEMENTATION

1. Assume a full replication scheme (i.e., update data dictionary accordingly).
2. Force the method for delegate replica selection in the client side (line 6 in Algorithm 3.2) to always return the identifier of the primary copy.
3. Allow the client to select a random replica for read-only transactions (line 11 in Algorithm 3.2) instead of issuing them to the primary copy.
4. It is not necessary to update the RS nor the WS to all replicas for every $read$ or $write$ operation.
5. The 2-Phase Locking (2PL) protocol for $commit$ operations (e.g., Figure 3.10) must be solely resolved locally at the primary copy (i.e., vc_i and dc_i/dnc_i messages in Figure 3.13 can be removed).

1SR LAZY REPLICATION PROTOCOL IMPLEMENTATION

1. Return acknowledge messages ($r_i(x_k)$ and $w_i(x_k)$) as soon as the delegate replica receives the $read/write$ operations instead of waiting for participants reply (see Figures 3.11 and 3.12).

Note that according to the guarantees stated in Equation 3.2.4 and considering that a server will not process more than one operation at a time (see Algorithm 3.5), this lazy replication implementation will also produce 1SR histories.

This concludes the implementation of classic concurrency control and replication protocols for transactional distributed databases. In the next section, we will evaluate the performance of these protocols in order to justify their scalability limitations.

3.4 PERFORMANCE EVALUATION

Concurrency control and replication protocols have a considerable impact on the reduced scalability of transactional distributed databases [Gray et al., 1996]. In the literature, several proposals have been proposed to mitigate this effect under some concrete circumstances [Brewer, 2000; Patiño-Martínez et al., 2005; Serrano et al., 2007; van Renesse and Guerraoui, 2010]. Nonetheless, assessing to what extent new techniques outperform the existing ones becomes an arduous task [Kemme, 2000; Wiesmann and Schiper, 2005]. The purpose of this section is to find out the source of scalability limitations in transactional databases, which extends the exhaustive performance analyses conducted by other authors in the past [Pedone, 1999; Kemme, 2000; Plattner, 2006].

Experiments	Exp. 1	Exp. 2	Exp. 3	Exp. 4
Number of servers	5–225	12	75	100
Transaction type	Short	Medium	Long	Varying
Load	250 tps	Varying	100 tps	75 tps
Update Transactions	20%	10%	30%	20%
Communication cost	Low	Low	High	Medium
Replication degree	100%	100%	Varying	100%

Table 3.1: Parameters settings for each experiment.

This section presents four experiment suites. Table 3.1 provides a summary of the parameters settings for each experiment. All experiments have been run 100 times to reach statistically significant values. When possible, these parameters have been set closed to the ones proposed by previous works [Kemme, 2000; Serrano et al., 2007] in order to obtain comparable results and, thus, validate our approach. The first experiment provides a general scalability analysis comparing the NDCC, BOCC, and BRP as the number of nodes increase, which extends the work of Kemme [2000] up to 225 servers. The second experiment analyzes the performance of each algorithm in terms of abort rate and throughput as the number of transactions per second issued by every client increase, which is very similar to the analytical experiment carried by Serrano et al. [2007]. The third experiment evaluates the effects of the replication degree of each protocol for a mixed workload consisting of update transactions and queries, which supports the work conducted in Gray et al. [1996]. Finally, the fourth experiment analyzes the effect of the transactions size over the abort rate and throughput, which envisages the fundamentals of cloud-based databases.

3.4.1 *Experiment I: Scalability*

This experiment provides a general scalability analysis to portrait the scale up abilities of each protocol as the number of servers increase. It compares the response time and the abort rate of transactions in a full replicated scheme ranging from 5 to 225 servers. Note that the implementation of the NDCC, BOCC, and BRP algorithms with full replication does not include the RS broadcast for read operations. For the sake of this experiment, we have chosen (1) a fast communications network (i.e., low communications overhead Kemme [2000]), (2) a fixed workload of 20% short update transactions and 80% short read-only transactions, and (3) an inter-arrival time of 4 ms (see `DelayBegin` and `DelayOP` in Section 3.2.2.1) per node as done in Kemme [2000], which leads to a constant system load of 250 tps.

This workload lets us observe how the system performance is progressively degraded. Such degradation comes from the fact that the more replicas are involved in the replication process, the more messages have to be exchanged, which increases the communications network stress [Gray et al., 1996].

Figure 3.15 depicts the response time and abort rate for NDCC, BOCC, and BRP as the number of sites in the system increases from 5 to 225. We observe that BRP behaves better than the BOCC as the number of replicas rises. The abort rate of BRP is slightly better than BOCC; such improvement becomes more relevant as the number of sites

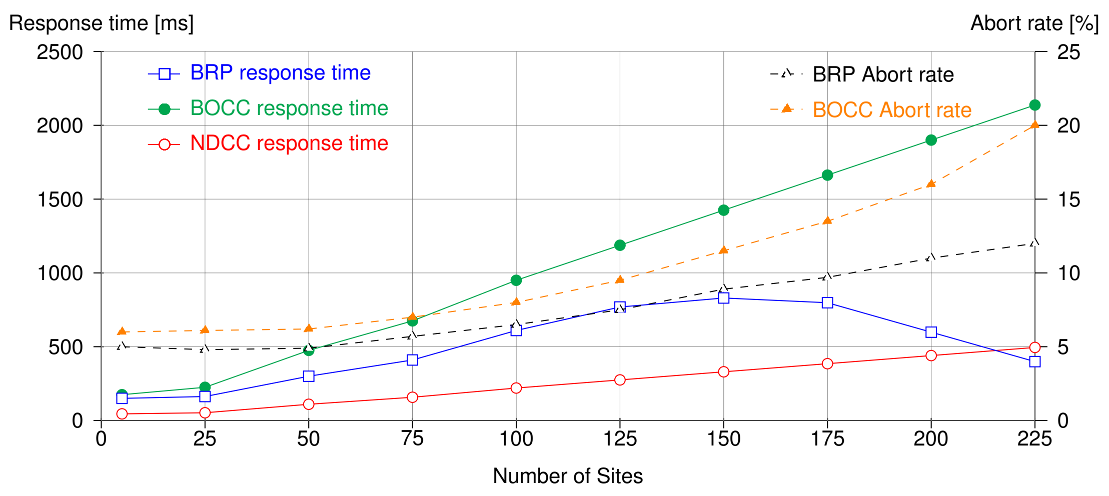


Figure 3.15: Experiment I. Response time and abort rate for different number of sites.

increase. Note that the abort rate for NDCC is zero since conflicts are not verified, which permits all transactions to commit. For 150 nodes and higher, we observe that the transactions response time when using BRP plunges. Finally, for 190 replicas and more, the transactions response time when using BRP is lower than the ideal case portrayed by the NDCC.

Analysis. Killing those transactions that are going to abort as soon as possible—as done by BRP—contributes to keep the transactions response time under reasonable values, which improves the system performance considerably. When the number of sites is high (i.e., more than 120) the conflict probability rises due to the fact that transactions remain more time active in the system, which increases the abort rate. This situation forces the system to abort many transactions that might be waiting for this large number of replicas to be synchronized. As several transactions are prematurely aborted by the BRP algorithm, the average response time is reduced—even reaching lower values than NDCC—as the number of replicas rise. Overall, we have found that these protocols scale fairly well up to 100 servers, which agrees with the work conducted by Kemme [2000]. However, for more than 100 replicas, the network starts behaving as a bottleneck due to the large number of exchanged messages between replicas and, thus, the system performance rapidly degrades. Also, we have observed that this effect is emphasized if a higher load is submitted to the system (e.g., with 1000 tps the scale out threshold would be at 15 replicas rather than at 100).

3.4.2 *Experiment II: Throughput*

This experiment analyzes the performance of each algorithm in terms of abort rate and throughput as the database load increases, which is very similar to the work carried by Serrano et al. [2007]. Hence, to obtain comparable results with Serrano et al. [2007] and validate our approach, we selected a workload that models the TPC-W benchmark [Nambiar et al., 2010]. Also, we set a scenario with 12 replicas that ran medium sized transactions (10% update and 90% read-only) with a low communication cost overhead

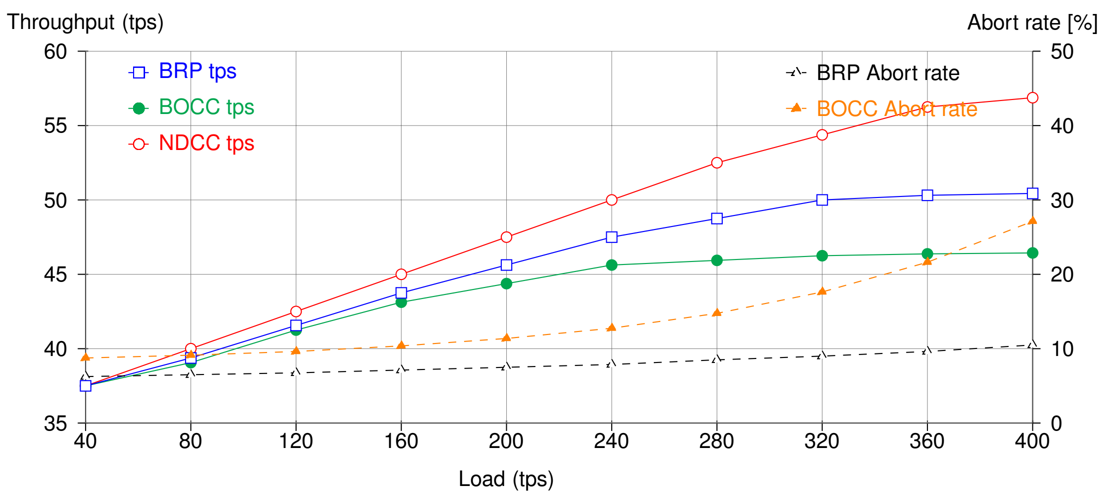


Figure 3.16: Experiment II. Throughput and abort rate for different loads.

[Kemme, 2000]. As shown by Serrano et al. [2007], fully replicated databases with this configuration typically collapse for more than 400 tps. Therefore, we selected a varying load that ranges from 40 tps to 400 tps, which enables a detailed behavior analysis for the NDCC, BOCC, and BRP algorithms. To conduct this experiment, we also modified clients' behavior (see Algorithm 3.2) to implement the synchronous behavior requested by the TPC-W benchmark [Serrano et al., 2007; Nambiar et al., 2010]. This modification forces the client to wait for the operation acknowledge from the server before issuing the next operation.

Figure 3.16 depicts the response time and abort rate for NDCC, BOCC, and BRP as the number of issued transactions per second increase. We observe that BOCC reaches the collapse threshold earlier than BRP. The abort rate of BRP remains pretty stable for all loads while the abort rate of BOCC rises as soon as the database collapses. The database collapses for more than 380 tps, 320 tps and 240 tps when running the NDCC, BRP, and BOCC algorithms respectively.

Analysis. As now the system behaves as an open system [Schroeder et al., 2006] rather than a partly-open one (i.e., clients are no longer forced to issue operations at a fixed frequency), the performance has considerably dropped compared to the first experiment. In addition, the obtained throughput is lower than the one reached by Serrano et al. [2007] because our NDCC, BOCC, and BRP implementations provide a serializable—rather than snapshot—isolation level, which forces them to exchange more and bigger messages. Nonetheless, the asymptotic behavior shown in Figure 3.16 is similar to the one depicted by Serrano et al. [2007], which supports the faithfulness of our approach. Additionally, we see that reducing the number of update transactions lets the system to keep the abort ratio under low values. Overall, the throughput rises linearly until at a certain point where servers are unable to process more operations and, thus, the network queues rise. This situation increases the transactions response time and, consequently, the conflict probability and abort rate as shown in Experiment I.

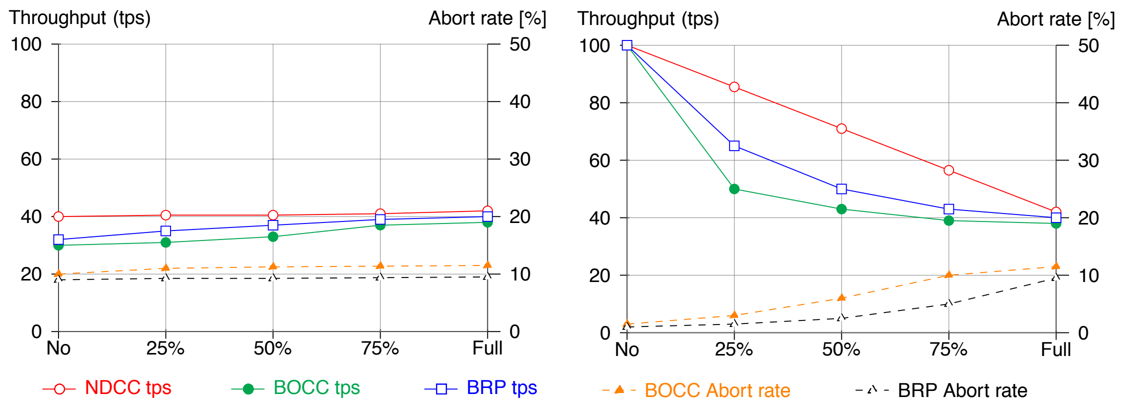


Figure 3.17: Experiment III. Throughput and abort rate for different replication degrees. Multi-partition transactions on left and no multi-partition transactions on right.

3.4.3 *Experiment III: Replication*

This experiment aims to further understand the effects of replication in a distributed database [Gray et al., 1996; Pedone et al., 2000]. Therefore, it analyzes the performance of NDCC, BOCC, and BRP algorithms in terms of abort rate and throughput as the number of replicas per object varies, which is an alternative approach to the work conducted by Armendáriz-Iñigo et al. [2008]; Paz et al. [2010b]; Peluso et al. [2012]. To emphasize the consequences of having the same object replicated in different nodes, we chose a (1) low communications network (i.e., high communications overhead Kemme [2000]), (2) fixed workload of 30% long update transactions and 70% long read-only transactions, (3) 75 servers layout, and (4) 100 tps for the system load. This configuration allows us to keep the database on an intermediate state where it is neither over saturated nor oversized—as shown in Experiment I and Experiment II—and emphasize the effects of replication. To conduct this experiment, we restored clients' module to their initial implementation depicted in Algorithm 3.2, which leaves the database as a partly-open system again. In addition, by properly adjusting the hot-spot area parameters in the workload generator (see Section 3.2.1) we synthesized two different workloads:

1. Workload A: It is built upon a random generated object access pattern. That is, the hot-spot area embraces all objects. Hence, multi-partition transactions (i.e., the delegate replica does not own the requested transaction's object and has to forward the request to another site) may often occur.

2. Workload B: It is built upon a coherent object access pattern. That is, hot-spot hit frequency is set to 100% and every client is assigned to a fixed hot-spot area that embraces all the objects owned by the client's delegate replica. Hence, no multi-partition transactions occur.

Figure 3.17 depicts the throughput and abort rate for NDCC, BOCC, and BRP as the replication degree (i.e., ratio of servers that own a copy of every object) increases. Workload A results are plotted on the left side and Workload B on the right side in Figure 3.17. We observe that the throughput and abort rate remain pretty steady for Workload A. For Workload B, we observe that the more replicas are added to an object, the higher abort rate is obtained and the less throughput is reached.

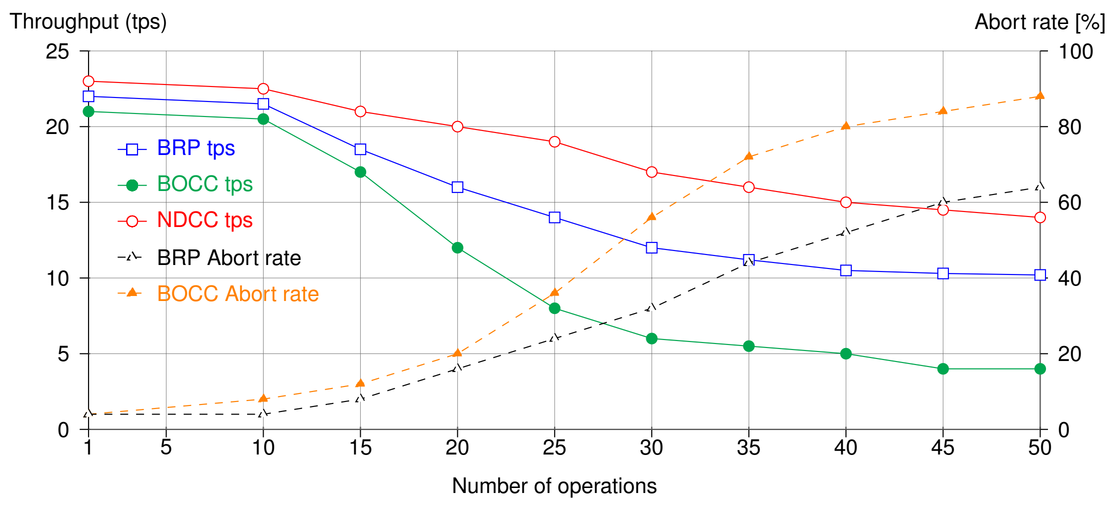


Figure 3.18: Experiment IV. Throughput and abort rate for different transaction sizes.

Analysis. Reducing the number of replicas should alleviate the number of synchronization messages in the network and, thus, increase the throughput at the expense of availability and fault tolerance [Brewer, 2012]. However, if the partitioning scheme is not carefully designed and, as a result, multi-partition transactions occur, the system is unable to experience such improvement as shown for Workload A. From the replication point of view, forwarding an operation request of an object that it is not owned locally, is equivalent to merge the local partition with the partition owned by the remote replica. Therefore, most extra replicas take part on the replication process and, thus, the benefits of partial replication are imperceptible as shown in Figure 3.17 for Workload A. Also, this effect is stressed by the serializable isolation level provided by our NDCC, BOCC, and BRP implementations [Serrano et al., 2007].

On the contrary, if no multi-partition transactions are allowed (Workload B), the system behaves in a more appealing way. When no replication is conducted, all servers are able to process the input load (i.e., 100 tps). The more replicas are included to the system, the more synchronization messages are sent, the more time transactions remain active, and, thus the higher conflict probability. Note that we observe that when no replication is selected for Workload B, there are still few transactions that are unable to commit. This comes from the fact that there is some local concurrency at every site. Overall, these results support the work carried by Gray et al. [1996]; Brewer [2012].

3.4.4 *Experiment IV: Transaction size*

This experiment explores the effects of varying the number of operations contained inside a transaction besides N_i (i.e., transaction size) in terms of throughput and abort rate. Kemme [2000] uses two fixed values for transaction length (i.e., 10 operations in sort transactions, 30 operations in long and query transactions) and some performance enhancements are slightly envisaged. Apparently, the less operations a transactions has, the less time will last the transaction and, thus, the less conflict probability might be obtained, which would result in a lower abort rate as shown in Experiment I. However, if transactions were forced to be as short as possible, application designers would have to carry on reasonably complex software developments to handle those actions that inevitably require long transactions. Hence, a new trade-off between the transaction length and the application logic complexity appears. To quantify the benefits of reducing the transaction size, we analyzed the behavior of NDCC, BOCC, and BRP when the transaction size ranges from 1 to 50 operations. For the sake of this experiment, we chose a (1) medium communications cost overhead, (2) fixed workload of 20% update transactions and 80% read-only transactions, and (3) 75 tps for the system load.

Figure 3.18 depicts the throughput and abort rate for NDCC, BOCC, and BRP as the number of operations per transaction increases from 1 to 50. We observe that there are not considerable performance differences between BRP and BOCC when the number of operations is low. For high values of number of operations throughput plunges and abort rate rockets for both, BRP and BOCC. Reducing the number of operations of a transactions from 50 to 5 (i.e., 90%) dramatically reduces the abort rate and rises throughput for more than 400% for BOCC and 200% for BRP.

Analysis. Indeed, when the number of operations is low transactions are less time active on the system, which (1) makes conflict situations rare and (2) reduces the size of network messages. For 20 operations and beyond, abort rate starts rising and performance decreases. We found that although the BRP tolerates this situation better than the BOCC, both of them reach unacceptable abort rates (higher than 50%) for a high number of operations.

After conducting these four experiments, we conclude that (1) the proposed database simulator obtains performance values that are comparable with the existing literature, (2) including more servers to a transactional database does not bring a greater performance (i.e., reduced scalability), (3) partial replication (also referred to as data partitioned systems) brings great benefits in terms of performance as long as no multi-partition transactions are allowed, (4) the communication network plays a crucial role on limiting the database performance, (5) it makes sense to reduce the number of operations per transaction to boost system performance, and, thus, system scalability. The following chapter takes advantage of these lessons and addresses a new type of databases referred to as cloud databases.

Contribution.

1. Description of the simulation environment for distributed databases.
2. Implementation of the NDCC, BOCC, and BRP algorithms for partial replication.
3. Performance evaluation of concurrency control and replication protocols.
4. Discussion on the limitations of transactional systems.

4 CLOUD STORAGE: BOOSTING DATA STORAGE SCALABILITY

Summary. Traditional transactional storage systems fail to efficiently store and retrieve vast amounts of information—arisen from data-driven modern applications—due to their intrinsically limited scalability and poor tolerance to highly dynamic environments (e.g., servers joining or leaving the database, addition or deletion of data columns/rows). In the recent years a new way to exploit data, coined as cloud storage, has emerged as an alternative to overcome these drawbacks by (1) drastically reducing the transactions length, (2) moving computation units to data facilities, and (3) supplying hardware resources on-demand to fulfill the ever-changing performance requirements of each application. This chapter reviews the fundamentals of this new storage paradigm and discusses its benefits under a use case scenario[3].

"Engineers should stand on the shoulders of those who went before, rather than on their toes"
— Michael Stonebraker, 2010.

4.1 INTRODUCTION

Despite the latest efforts on designing and developing efficient concurrency control and replication algorithms, scalability is still a challenge in traditional transactional databases [Brewer, 2000; DeCandia et al., 2007; Paz et al., 2010a; Brewer, 2012]. Rapid advances in technology and storage capacity have rocketed the volumes of data arisen from internal and external business processes, which has raised data management to a crucial component in many data-driven applications [Lynch, 2008]. Additionally, the concept of data management has evolved and, currently, not only refers to data storage but also to computation and data aggregation. Indeed, the performance limitations suffered by traditional transactional databases when facing large data volumes (i.e., Big Data [Lynch, 2008])—already exhibited in Chapter 3—have definitely driven practitioners to rethink this kind of systems [Jacobs, 2009; Stonebraker et al., 2010].

Nowadays, storage and communication networks have evolved in such sense that it is actually faster and cheaper to ship a set of modern hard drives via overnight delivery service, than transferring all their contents over a regular communications network [Armbrust et al., 2009]. Note that this breaks away from the classic idea of moving data to computation historically pursued by transactional databases—where a delegate replica collected all data from different sites (see Chapter 2)—and proposes to move computation units to data instead.

3 An earlier abridged version of the work reported in this chapter was published as the paper entitled "Cloud computing keeps financial metrics computation simple" in the proceedings of the 6th International Conference on Software and Data Technologies (ICSOFT 2011).

This situation has led to a new exploitation model of hardware resources, coined as cloud storage, which is targeted to overcome the limitations of transactional databases when addressing Big Data [Agrawal et al., 2011]. This chapter reviews the fundamentals of cloud storage, seen as an *evolution* of classic transactional databases, and explores its benefits through a use case scenario using Apache Hadoop [White, 2011]—one of the most popular state-of-the-art cloud storage frameworks.

4.2 CLOUD STORAGE FUNDAMENTALS

Formally, cloud storage has emerged as a solution to address new concerns derived from managing Big Data in modern applications and devoted to provide both resources on demand (also referred to as elasticity) and virtually infinite scalability [Kraska et al., 2009b; Armbrust et al., 2009].

On the one hand, provisioning resources on demand is an appealing feature for any data-driven application (e.g., Twitter, Facebook, Amazon, Google, Dropbox) since it may result in a considerable reduction of capital and operational expenditures. In fact, the more users each application has to serve, the more resources can be dynamically added, which is against of the classic Kendall's queuing theory used to statically (over)dimension distributed systems [Kendall, 1953]. Accordingly, when the number of users or workload decreases, some resources can be turned off, which may result in an increase of hardware life cycle and energy saving. On the other hand, high scalability becomes a mandatory requirement when facing an ever rising number of users and applications that exploit the same physical resources [Das, 2011].

Nevertheless, in order to provide these two features some other system characteristics must be relaxed [Wei et al., 2012]. For instance, as shown in Chapter 3, transactional databases performance can be boosted under some concrete circumstances (e.g., reducing transactions length, properly partitioning data, reducing the isolation level) in the detriment of other properties (e.g., relational algebra, application design or upper-layer software features). This effect was formalized by Brewer's theorem [Brewer, 2000; Gilbert and Lynch, 2002], which states that to efficiently deal with a deluge of data spread over thousands of servers it is necessary to come up with a reasonable trade-off between data consistency, availability, and network partitioning properties [Brewer, 2012].

In the rush to obtain highly scalable systems and push databases performance to the top, early cloud-based systems drastically reduced the scope of another important feature from classic databases: transactions. Despite their benefits in data management at the application layer, transactions usage significantly degrades the system performance (see Chapter 3) by (1) locking objects that prevent subsequent operations from progressing, (2) generating heavy messages with long WSs (and also RSs depending on the selected isolation level) that saturate the network, and (3) running expensive concurrency control algorithms to deal with actions that span a high number of servers. Therefore, limiting transactions' size to one operation (also referred to as key-value stores) greatly reduced their associated burden.

Hence, current cloud-based storage systems (e.g., Amazon S3 [Palankar et al., 2008], Dynamo [DeCandia et al., 2007], Yahoo! [Cooper et al., 2009] or Hadoop [White, 2011]) are able to offer (virtually) infinite scalability and availability at a low cost [Ghemawat et al., 2003; Kraska et al., 2009b] by smartly exploiting these principles (i.e., usage key-value pairs and selecting a convenient trade-off between consistency, availability, and network partition tolerance). Nevertheless, this adds more complexity to high-level applications which have now to be aware of (and deal with) possible data issues arisen at the storage layer [Aguilera et al., 2009; Levandoski et al., 2011; Das et al., 2010b; Corbett et al., 2012; Wei et al., 2012].

This section (1) further elaborates on the properties of transactional databases that have been relaxed to meet the cloud paradigm, (2) details a generic system architecture to enable system elasticity and scalability, and (3) reviews the most popular up-to-date cloud storage solutions.

4.2.1 *Limiting classic transactional database features*

Big Data storage systems must be (1) elastic (i.e., resources can be added or removed at will in order to adapt to user demands), (2) scalable (i.e., no performance fluctuations should happen when the number of users and/or resources changes), (3) available [Armbrust et al., 2010] (i.e., service disconnections never occur), and (4) cost-effective [Kaushik et al., 2010] (i.e., pay-as-you-go basis) to successfully meet the resource exploitation model stated by the cloud storage paradigm. As shown in Chapter 3, classic distributed transactional databases generally fail at fulfilling all these requirements at a time. To address this concern, cloud storage repositories (often over-characterized to as cloud databases) have limited the following data properties and left their management to upper-layer applications.

Consistency. Keeping strong data consistency [Mosberger, 1993; Steinke and Nutt, 2004] in a transactional distributed system is, a priori, expensive [Krikellas et al., 2010] because all updated objects involved in a transaction must be locked until they reach the same state (see Chapter 2). As shown in Chapter 3, this potentially prevents concurrent transactions to progress and, thus, considerably limits the system throughput [Gray et al., 1996]. Using weaker consistency models [Vogels, 2009] can greatly contribute to boost the system throughput at the cost of solving data conflicts at the application layer [DeCandia et al., 2007; Adya, 1999]. The most popular weak consistency model in cloud storage repositories is eventual consistency [Vogels, 2009], which guarantees that (1) every replica sees updates in the same order and (2) all replicas will eventually converge to the same state if no new updates are applied—considerably reducing the overhead associated to consistency management.

Data models. Exploiting relational data schemes through transactions, as done by classic distributed databases, results in a powerful way to store and retrieve information from the application layer. However, these data models are prone to keep data logically tied together, which prevents the distributed system from scaling horizontally. In the last few years, an alternative approach referred to as NoSQL has emerged to increase the scalability of the distributed systems storage layer at the cost of losing the advantages of relational structures and algebra [Cattell, 2010; Stonebraker, 2010]. Specifically, NoSQL, which is used by several cloud storage repositories [DeCandia et al., 2007; Chang et al., 2008; Carstoiu et al., 2010; Cattell, 2010], consists of (1) decoupling data relations, (2) simplifying data models (i.e., key-value pairs), and (3) using simple data retrieval and appending operators (e.g., `get(key)`, `put(key,value)`) [White, 2011].

Transactions. Ensuring ACID properties over a closed set of distributed data objects becomes a straight-forward task as long as usage of transactions is allowed (see Chapter 2). However, transactions considerably limit the performance and scalability of distributed databases due to the overhead associated to data replication (i.e., large WS to be applied over a large number of replicas) and concurrency control (i.e., several transactions competing for the same object set) [Gray et al., 1996]. Therefore cloud storage repositories [Ghemawat et al., 2003; Brantner et al., 2008; White, 2011] further decouple data objects and boost database scalability by reducing the transactions size to one operation (i.e., key-value stores)—which is also a consequence of exploiting the NoSQL paradigm.

Data storage facilities. Storing data to persistent storage (also referred to as secondary memory) ensures durability and eases recovery in case of faulty behaviors [Cristian, 1991; Castro and Liskov, 2002; Kapitza et al., 2012]. However, secondary memory usually behaves as a bottleneck since it is often much slower than main memory [Aguilera et al., 2009; Jones et al., 2010]. Thus, cloud storage repositories [DeCandia et al., 2007; White, 2011] tend to store as much data as possible into main memory and periodically flush data to disk. Additionally, some cloud repositories choose to limit the negative consequences of using persistent storage—while keeping reasonable fault tolerance and durability facilities—by limiting the scope of disk operations (e.g., batch writing, Write Once and Read Many (WORM) policies, fixed-size data blocks) [Ghemawat et al., 2003; Chang et al., 2008; White, 2011].

Overall, cloud storage repositories [Ghemawat et al., 2003; DeCandia et al., 2007; Chang et al., 2008; Lakshman and Malik, 2010; White, 2011] can be seen as non-relational databases: plain databases with no special features such as fast interfaces or advanced concurrency control algorithms, where data are just stored in a non-normalized scheme to boost availability, scalability, and elasticity.

4.2.2 *Generic architecture layout*

As data properties and database features shall be greatly reduced when addressing Big Data and cloud-based storage services, it is no longer necessary to maintain all the burden associated to transactions as classic DBMSs do (see Chapter 2 and Chapter 3).

The first well-known approach in the literature to this commitment was proposed by Ghemawat et al. [2003] and unveiled the internals of the Google File System (GFS). This proposal has become a reference model for the design of the wide majority of current cloud storage systems [Chang et al., 2008; Cooper et al., 2008; Dean and Ghemawat, 2010; Das et al., 2010b; White, 2011; Corbett et al., 2012].

The goal of GFS was to build a highly scalable storage layer able to (1) face the new challenges posed by Big Data and, thus, deliver high aggregate performance to a vast number of users (i.e., massively exploiting data locality), (2) using inexpensive commodity hardware to store all data, and (3) provide fault tolerance (i.e., elastic provisioning of hardware resources) [Ghemawat et al., 2003]. In this regard, the classic architecture proposed used in classic databases Bernstein et al. [1987] was completely redesigned: the distributed system was no longer the composition of several centralized systems and data management was physically decoupled from data storage. Therefore, the GFS defines three logical entities which are described in what follows.

GFS chunkserver. It stores data objects, conducts replication duties using chain replication [van Renesse and Schneider, 2004], and executes client data queries. As no transactions run in GFS, all queries are made in terms of key-value pairs, which means that a GFS client is only allowed to issue either a `put(key,value)` to write data or a `get(key,value)` to retrieve data—following a WORM scheme. Also, to further boost system performance [Dean and Ghemawat, 2010] and ease data objects management [White, 2011] GFS chunkservers are forced to deal with fixed-size and user-defined data blocks (also referred to as data chunks), which has a considerable impact on the design and development of end-user applications [Ghemawat et al., 2003].

GFS master. It stores and manages all the information related to every data object physical location (also referred to as metadata) and, thus, it is responsible for assigning a delegate site (i.e., GFS chunkserver) for every client request. Therefore, a client needs to ask the GFS master for the location of an object prior retrieving it. In this regard, this entity also behaves as a load balancer since (1) it is aware of all the actions conducted at each GFS chunkserver, (2) periodically polls GFS chunkserver about their status, and (3) decides to move data chunks among GFS chunkservers to exploit data locality. It is worth mentioning that GFS master does not lock any data object, which leads the GFS to offer eventual consistency [Vogels, 2009].

GFS client. It links end-user applications with the GFS. Specifically, it communicates the client with the (1) GFS master for metadata related operations and (2) GFS chunkservers for data-bearing operations. Such interactions, next to the replication and chunk building, are made transparent for the application layer through an Application Programming Interface (API).

As a result, two types of data flows between these entities are distinguished:

1. Data messages containing data objects that are exchanged between the GFS clients and the GFS chunkservers.

2. Control messages containing (1) object locations and indexing information that are exchanged between the GFS clients and the GFS master and (2) status information and maintenance instructions that are exchanged between the GFS master and GFS chunkservers [Ghemawat et al., 2003].

Therefore, when new servers need to be added or removed from the system to meet a specific workload—as demanded by the cloud philosophy—, it is only necessary to notify the GFS master which chunkservers are (no longer) available. This action results on a small number of control messages and has few impact on the heavy data communication traffic.

The goodnesses and benefits portrayed by Ghemawat et al. [2003] about GFS have driven the community to abstract it into a general reference model to develop, modify, and benchmark alternative solutions [Chang et al., 2008; Cooper et al., 2008; Dean and Ghemawat, 2010; Das et al., 2010b; White, 2011; Corbett et al., 2012]. This abstract model is depicted in Figure 4.1 and it contains a generalization of the entities defined by the GFS: storage facilities, metadata manager, and clients. These entities are detailed in what follows.

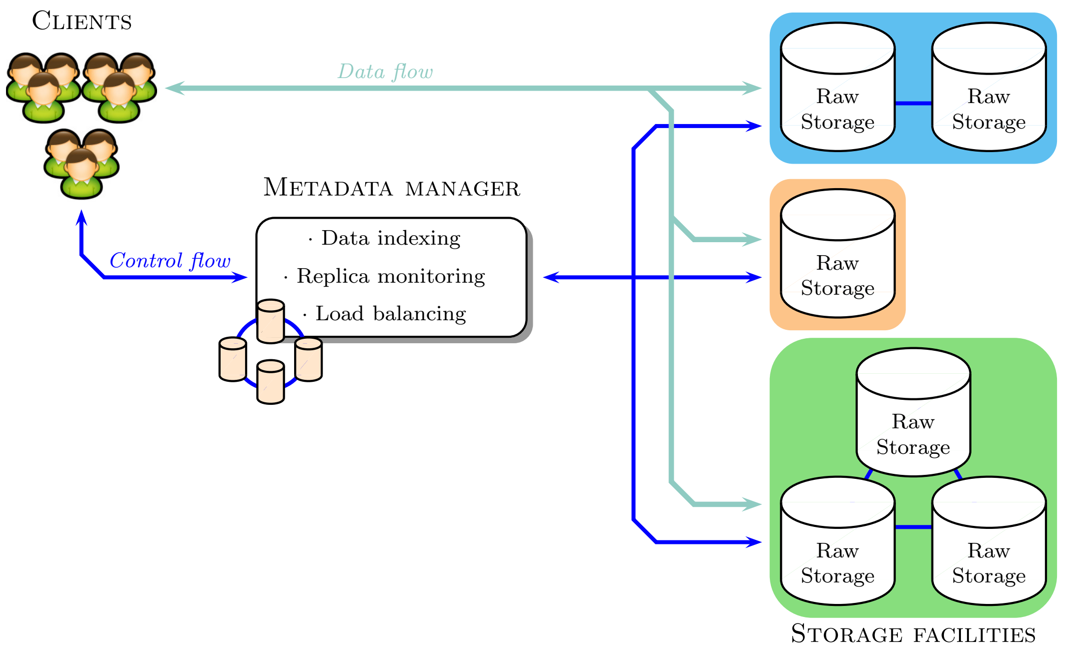


Figure 4.1: Generic reference model architecture for cloud storage.

Storage facilities. Similar to GFS chunkservers. Storage servers can be grouped (colored areas in Figure 4.1) either logically—by application—or physically—by rack—to ease the replication process [Gray et al., 1996], improve the fault tolerance [White, 2011], and support multi tenancy [Das, 2011]. Also, they allow to further exploit data locality in order to reduce the number of data messages that cross the whole network.

Metadata manager. Similar to GFS master. In order to prevent the system from the single point of failure issue, the metadata manager entity can be deployed over a small set of servers. Hence, metadata is fully replicated on each one of these servers, which alleviates their individual load and eases the recovery process. Also, it manages and monitors storage facilities (blue arrows in Figure 4.1).

Client. Similar to GFS client. It asks the metadata manager to which storage facility has to forward its requests (blue arrows in Figure 4.1) and interacts with storage facilities to store and retrieve data (green arrows in Figure 4.1).

This modular design, that basically consists of splitting data management duties from storage facilities, has served as a basis to build highly scalable data storage repositories and meet some industry requirements as shown in what follows.

4.2.3 *Existing cloud storage platforms*

Indeed, there are several approaches in the industry that have taken this generic reference model as a starting point for their contributions. This section reviews the most relevant examples in the literature and highlights different strategies to obtain a highly scalable repository: Apache Hadoop [White, 2011], ElasTraS [Das et al., 2010b], Amazon Dynamo [DeCandia et al., 2007], Yahoo! PNUTS [Cooper et al., 2008], and Google Spanner [Corbett et al., 2012].

Apache Hadoop framework. It is an open-source software stack (also referred to as Apache Hadoop ecosystem) specially designed to handle massive amounts of data (i.e., Big Data); it provides a storage layer on its bottom and a set of data processing tools on its top. On the storage side, it uses a public implementation of the GFS named HDFS. On the processing side, it includes several frameworks to conduct different data-related tasks such as: Hive (used by Facebook or Netflix) for data summarization and analysis through an SQL-like interface [Thusoo et al., 2009], Pig (used by Twitter or LinkedIn) for analyzing and evaluating large data sets using parallelization, Chukwa for large-scale log collection and analysis, or HBase (used by the—recently deprecated—messaging platform of Facebook) for implementing NoSQL distributed databases [Carstoiu et al., 2010; Harter et al., 2014]. Also, the Apache Hadoop ecosystem includes MapReduce [Dean and Ghemawat, 2010]: a middleware that allows practitioners (and applications from the Apache Hadoop stack as well) to easily run parallel computing tasks while abstracting them from the burden associated to distributed systems (e.g., determining data objects locations, synchronizing distributed tasks, reissuing failed tasks, etc.). It is worth considering that all these applications built on top of the HDFS storage layer typically inherit all its benefits—and limitations—regarding data storage (i.e., fault tolerance, elasticity, scalability, availability, and eventual consistency).

Overall, Apache Hadoop provides a highly scalable and available architecture to store vast amounts of data and provides a set of software tools to conduct Big Data related tasks.

ElasTraS. It provides a transactional database for the cloud inspired by the Apache Hadoop ecosystem (i.e., splitting the system into a storage layer and an application layer). In this regard, it uses the HDFS to have a highly scalable persistent storage system and deploys on its top a set of entities to manage transactions and consistency [Das, 2011]. Additionally, to avoid limiting the scalability and elasticity of the storage layer because of transactions execution (see Chapter 3), it partitions the database at the schema level (although very few hints are provided on how to obtain partitions). As a result, it is able to enhance the weak consistency model featured by the HDFS and provide strong consistency inside every partition (a perfect partitioning scheme [Sancho-Asensio et al., 2014] is assumed and, thus, no multi-partition transactions are addressed by ElasTraS).

Overall, ElasTraS takes advantage of the new features provided by cloud-based approaches to overcome the limitations of traditional RDBMSs in terms of availability, scalability, and elasticity.

Amazon Dynamo. It provides a persistent data storage service—based on key-value pairs—to the services hosted by Amazon. It uses a variant of eventual consistency [Vogels, 2009] in which ordering properties are not validated in order to further boost availability (i.e., inconsistent updates are not blocked as long as the client which later reads their associated values is able to reconcile them). That means that a single key will map onto the union of values resulted from inconsistent updates. Note that in the specific case of Amazon (on-line shopping cart), this assumption has reasonably few impact from the user perspective since (1) additions to cart never get lost and (2) users are typically fine with correcting errors by themselves when an object they deleted on the past still appears on the cart. Nevertheless, Dynamo uses vector-clocks [Lamport, 1978a] combined with a quorum-based policy as a best-effort strategy to minimize the number of consistency corrections conducted by end-users.

Overall, Dynamo is a highly scalable and available distributed data storage system that slightly reduces the overhead associated to maintain eventual consistency in order to minimize latency (i.e., earlier consistent versions of an object are obtained) and, thus, improve end-users experience.

Yahoo! PNUTS. It provides a geographically distributed database service specifically designed to address the challenges posed by the Yahoo! applications (e.g., social web and advertising) in terms of scalability, slightly relaxed consistency guarantees, good response time for world-wide users, and high availability [Cooper et al., 2008]. From the architecture perspective, PNUTS uses a particularization of the model depicted in Figure 4.1: the entity of the metadata manager (now split as a set of Routers and Tablet Controllers) is distributed across geographically distant regions and uses a Message Broker [Cooper et al., 2008] to redirect data requests to each region. Also, it uses a primary backup approach combined with lazy replication (see Chapter 2) to boost availability and fault tolerance. From the data perspective, PNUTS allows storing relational data—rather than simple key-value pairs—in hashed tables exploiting write operations locality (which are inherent to the aforesaid applications) providing per-record consistency guarantees (i.e., all operations issued to the same record reach a 1SR isolation level but no ordering guarantees are provided between operations to different records).

Overall, PNUTS provides a highly scalable and available database (inheriting the elasticity features derived from using the scheme depicted in Figure 4.1) that is able to hold a richer data model (i.e., relational tables) in exchange of reducing the scope of transactions (i.e., single-record oriented transactions are only supported with 1SR) and, thus, obtaining a stronger consistency model than eventual consistency—but still weaker than strong consistency.

Google Spanner. It provides a geographically distributed database service with transactional facilities for F1: the Google's ad business platform [Shute et al., 2012]. Spanner is conceived as a highly scalable alternative—without giving up relational semantics—to (1) the traditional MySQL databases that needed time-consuming resharding tasks as the amount of data to be stored grew, (2) the write throughput limitations posed by the Google's semi-relational data store Megastore [Baker et al., 2011], and (3) the burden associated to handle complex and evolving schemas with strong consistency guarantees—actually Spanner offers external consistency [Lin, 1989]—in wide-area replication suffered by Google's key-value store BigTable [Chang et al., 2008]. To offer transactional support and feature external consistency—there are some modern applications, such as the social web, that need even stronger guarantees than 1SR—, Spanner shards multi-version data across many sets of Paxos state machines [Lamport, 2006] in data centers spread all over the world and synchronously replicates them using 2PL (see Chapter 2) to take advantage of data locality and obtain global availability [Corbett et al., 2012].

In addition, to get rid of the scalability limitations of Paxos and 2PL (see Chapter 3) it uses a TrueTime API—based on GPS references and atomic clocks—that allows Spanner timestamp transactions and obtain a reliable global clock with reasonably small uncertainty.

Spanner's architecture is also inspired by the reference model depicted in Figure 4.1: it names Spanservers to the storage facilities—that store data in an enhanced version of the GFS referred to as Colossus—and splits the metadata manager in a Location Proxy—used by clients to locate data stored at Spanservers—and a Zonemaster to assign and balance data to Spanservers. Additionally, it replicates this layout to several geographically distant areas and coordinates them (i.e., moves data across zones to exploit locality) through a Placement Driver.

Overall, Spanner promises to bring back the classic benefits of transactional distributed databases on a highly scalable and multi-versioned basis without sacrificing any critical property such as availability or consistency. To achieve that, it precisely timestamps—thanks to time devices directly attached to the hardware resources—transactions and uses fast communication networks to rapidly move data where it is needed and, thus, minimize the latency.

Unfortunately, despite the benefits announced by these cloud storage technologies when addressing specific real-world problems (e.g., social web, advertising, or online shopping carts), the only one that is available as a free open-source software is Apache Hadoop. Therefore, it has been selected to face a custom real-world problem involving a large amount of non-homogeneous data in order to (1) further understand its behavior and (2) get hands-on experience about its advantages and drawbacks.

4.3 USE CASE SCENARIO

We propose to conduct this experiment on the financial repository Sistema de Análisis de Balances Ibéricos (SABI) [Bureau van Dijk, 2010], which is considered an important research tool by many Spanish universities [Albino, 2008] and is widely used by private companies to perform market analysis.

This large-scale data repository is targeted to engage researchers in analyzing companies' efficiency [Kapelko and Rialp-Criado, 2009; Retolaza and San-Jose, 2008; Guzmán et al., 2009] in terms of indebtedness, availability of idle resources, or capital costs [Martínez-Campillo and Gago, 2009]. To perform such calculations and extract valuable information, researchers and users have to follow a two-step procedure: (1) search and filter the data and (2) analyze them with the aid of statistical tools.

Despite SABI constitutes an important financial information source in Spain, many companies do not properly fill all their associated fields (i.e., incorrect values or simply missing), leading to an incomplete data panel. This is due to the fact that most of these companies manually introduce the calculated values. So far, authors [Hernández-Cánovas and Martínez-Solano, 2010] have roughly solved this issue by excluding the rows that belong to companies with missing (or misleading) values. However, reducing the size of the sample set or even replacing missing values with means may bias the results in terms of accuracy.

This situation opens two challenges regarding the SABI repository storage and computation. On the one hand, existing MySQL solutions fail at efficiently handling such an amount of heterogeneous data (i.e., more than a million of records with different number of fields). On the other hand, filtering and correcting misleading values in such a volume of information is an arduous task.

Therefore, the purpose of this experiment is to use an open-source cloud storage tool to efficiently store and process (i.e., fix misleading values whenever it is possible) the large amounts of data associated to SABI repository. To this end, we have selected HDFS [White, 2011] to store data and its implementation of MapReduce [Dean and Ghemawat, 2010] for parallel data processing (i.e., there is a set of operations that when performed with the distributed computation paradigm may speed-up the calculation time).

The remainder of this section is organized as follows. Section 4.3.1 further details the aforementioned software modules that have been picked from the Apache Hadoop framework to conduct this experiment. Section 4.3.2 describes the SABI repository. Next, Section 4.3.3 details the implementation and presents some experimental results. Finally, Section 4.3.4 summarizes the main lessons extracted and outlines some future research lines.

4.3.1 *Apache Hadoop. Data storage and processing*

The specification of this experiment is subjected to two requirements, data must be (1) strictly consistent and (2) written once and read each time calculations are performed (i.e., it is unlikely that records are edited once they are entered to SABI). In addition to being open source and provide a good balance between consistency and availability, Apache Hadoop is written in Java and designed to offer portability across heterogeneous hardware and software platforms, which eases its deployment in heterogeneous hardware and, thus, suitable for this experiment.

For the sake of this experiment, we have selected the following software tools available from the Apache Hadoop framework:

- **Distributed file system HDFS.** Storage devices tend to be the bottleneck [Paz et al., 2010a] in many scenarios such as web services or intensive computing applications; scenarios where user queries and network communications are faster than writing and reading from disks (see Chapter 3). However, HDFS, due to its architecture, behaves as a distributed file system mounted at the user space which spreads and replicates data across all the storage servers in a scalable way making use of main memory as much as possible.

  In our case, HDFS is used as raw storage container which ensures consistency, scalability, fault tolerance, and replication for WORM data (i.e., SABI). Data are automatically split—the size of each data fragment is set by default to 64 MB though it can be adjusted to obtain different performances [Shafer et al., 2010]—and each partition is stored on different sites enabling parallel distributed computations.

- **MapReduce.** Typically, data stored in distributed file systems can be processed either (1) by centralized computing (i.e., by aggregating all remote data and then processing them at a central node) or (2) by distributed computing (i.e., by first processing locally stored data chunks on each node and then aggregating the partial results). The latter may perform better than the first one when calculations can be solved in parallel, since it takes the most of the computational resources of each distributed site and minimizes the network traffic.

In our case, the SABI repository can be easily split (i.e., horizontally partitioned) since there are no dependencies between rows. Additionally, the data partitioning task is automatically conducted by the HDFS, which greatly reduces the development efforts of this experiment. Therefore, MapReduce is used as a distributed computing paradigm that hides the internal distributed file system (i.e., HDFS) architecture and allows processing distributed data without knowing its exact location.

- **HBase.** From the end-user point of view, data stored at HDFS can be accessed from either (1) the command line interface which gives direct access to the distributed file system via `put` and `get` HDFS directives—suitable to perform small tests and check whether data have been stored correctly—or (2) from an upper-layer middleware such as HBase [Carstoiu et al., 2010]—extremely useful when large amounts of data have to be read, processed, and written back to the file system as new records.

  In our case, HBase allows us to access to the non-normalized data (also referred to as heterogeneous) stored on the distributed file system as if they were on a structured distributed database. Both its standard query language and the HDFS built-in facilities make easier to formulate queries and retrieve filtered data. Therefore, HBase is used as an external entry-point interface to store and retrieve SABI's data.

Overall, HBase and MapReduce both assure to provide an efficient way to access the distributed data stored in the HDFS without compromising reliability nor worrying about data partitioning [White, 2011]. While HBase is best suited for real-time read/write random access to very large data sets, MapReduce is suitable for performing complex operations with stored data without having any notions about the typical issues of the distributed systems such as concurrency control, replication schemes, fault tolerance and recoverability.

As the goal of this experiment consists of analyzing the whole SABI repository (i.e., sequential access instead of random access), MapReduce seems to be a promising framework to efficiently deal with it and HBase will be left to conduct unit tests.

4.3.2 *The SABI repository*

This section briefly describes SABI, stresses its relevance, and points out its main drawbacks which will be addressed with the proposed MapReduce approach.

SABI is distributed online in Spain by INFORMA [Informa, 2010] and consists of (1) a private repository that gathers data from 1998 until 2009 of more than 1.2 million Spanish and Portuguese firms and (2) a basic financial analysis system.

As any other conventional database, data stored at SABI can be accessed through different search criteria such as company name, tax identification number, location, business activity, employees, etc. However, SABI provides additional functionalities that allow users to (1) perform statistical and comparative analyses of companies taking into account different variables and different time basis, (2) obtain reports in either standard or personalized format, and (3) graphically visualize results from balance accounts, income statements, and other comparisons. Therefore, SABI's strength lies on its analytical tools applied to finance—users can follow financial progress, carry out credit analysis, conduct company comparisons, identify competitors, study companies' position in the market, detect potential partners, consider mergers and acquisitions—, marketing—users can perform strategic corporate planning, examine market situation, detect potential clients, elaborate market strategies—, and economics research—users can benefit from a research tool and teaching resource.

Although SABI has been widely used in many research works [Retolaza and San-Jose, 2008; Hernández-Cánovas and Martínez-Solano, 2010], some of these studies have reported the inconsistency of the repository and the presence of missing values which force to remove many items from the database and, as a consequence, shrink the set of samples.

To address this issue, the following subsections show how the SABI repository has been integrated to HDFS to be later processed with MapReduce in order to detect and correct errors while keeping the information consistent.

4.3.2.1 *SABI Data file format*

As HDFS is designed to work with plain text files, the aforementioned SABI repository is extracted to a single text file of 10.4 GB—which is not even manageable for some file systems. Hence, it is split up into years (from 2001 to 2008) obtaining eight text files of 1.3 GB each (note that HDFS will split each 1.3 GB file into 64 MB data chunks as mentioned in Section 4.3.1).

The first row of each file contains the header indicating the content of each field (e.g., the name, address, number of employees, etc.) and the remainder of the file contains information regarding each company (one per row). Each company entry is written in a fixed-size virtual rectangle which forces long fields (e.g., name) to be written in multiple physical lines as follows:

```
This is  This is  43  Another  And another
a field  another      field    one
         field
```

In order to ease the processing tasks, these files need to be preprocessed to (1) demarcate each field (up to now there is not a unique field separator such as white space or tabulator), (2) transform multiple line fields into single lines, and (3) fill up the empty fields by inserting '⋆'. Such preprocessing task can be trivially conducted using any scripting tool (e.g., awk or sed). Once the 8 files are preprocessed, these are loaded into a 6 nodes HDFS cluster (1 Namenode and 6 Datanodes) and, thus, ready to be processed.

4.3.2.2 *SABI Data file contents*

The main problem that everybody faces when trying to extract any statistics from the SABI repository is the mismatch between the different values contained in the companies' accounts. Once mismatches are identified, these entries have to be either removed from the data panel or inferred from other cells. Therefore, the goal of this

Metric	Operations required
Total balance	*Total liabilities*
Total assets	*Shareholder contribution receivable* + *Long-term investments* + *Deferred charges* + *Current assets*
Non-current assets	*Start-up costs* + *Intangible assets* + *Tangible assets* + *Financial assets* + *Long-term treasury stock* + *Due on long-term traffic*
Current assets	*Expenditure required by shareholders* + *Stocks* + *Debtors* + *Short term investments* + *Short-term treasury shares* + *Treasury* + *Accrual*
Total liabilities	*Equity* + *Revenue deferred* + *Provisions for liabilities and charges* + *Long-term creditors* + *Short-term creditors*
Equity	*Subscribed capital* + *Premium* + *Reservations and results for previous exercises* + *Income* + *Interim dividend paid during the year* + *Share for capital reduction*

Table 4.1: Verified metrics from the SABI repository.

experiment is to efficiently automatize the task of identifying and fixing or removing these mismatches.

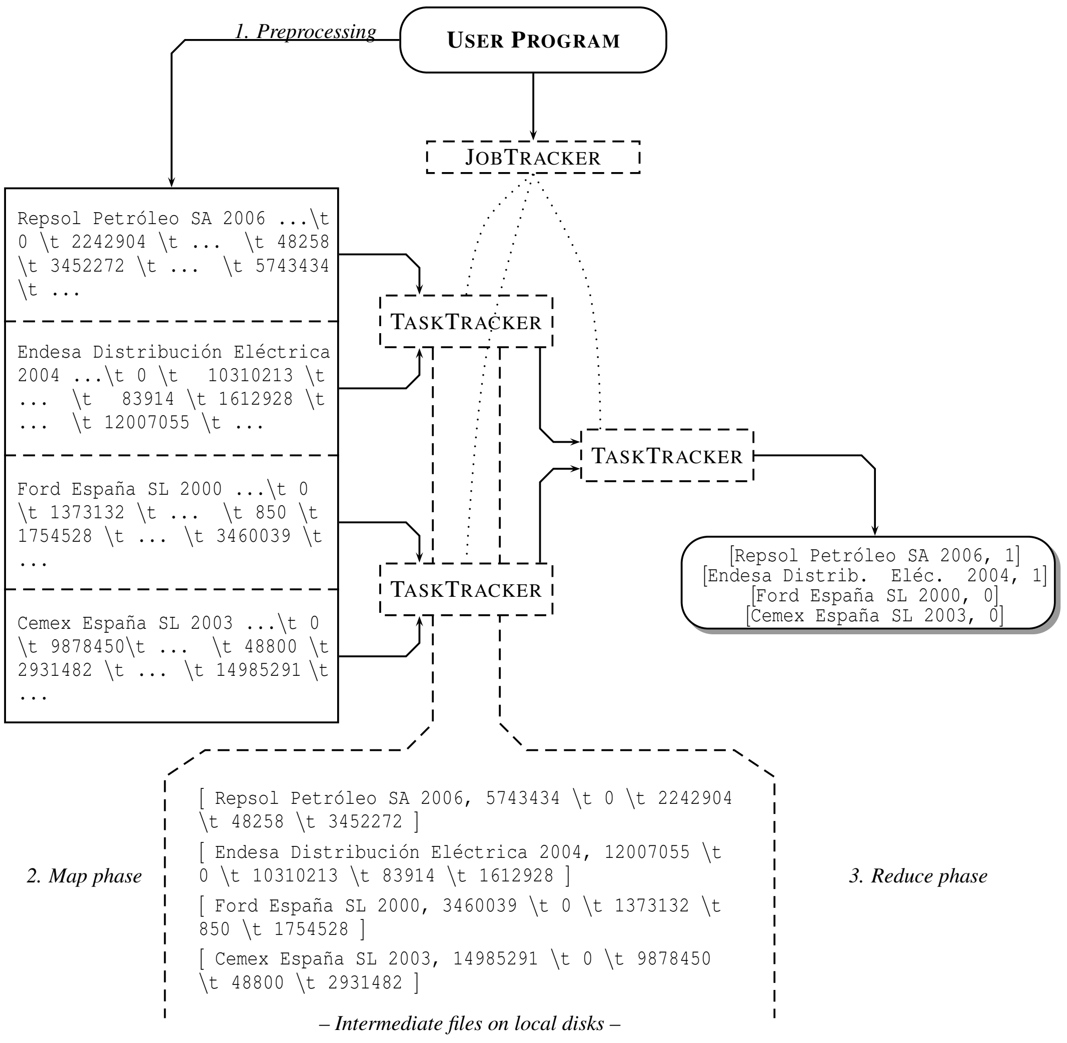
Table 4.1 shows some of the formulae used to conduct such verifications. For example, the *Total assets* value can be computed from the following items: *Shareholder contribution receivable*, *Long-term investments* (which is computed from the following items: *Start-up costs*, *Intangible assets*, *Tangible assets*, *Financial investments*, *Own stock* and *Long-term investments debtors*), *Deferred charges* (which is computed from the following items: *Shareholder contribution non-receivable*, *Debtors*, *Temporary financial investment*, *Short-term own stock*, *Liquid assets* and *Accrual adjustments*), and *Current assets*.

Hence, the *Total assets* field can be checked from such other fields of the same entry. In this specific case, if there is a field with incorrect data (either the final *Total assets* value or any of the others) the full entry will have to be removed since it is not possible to determine which is the wrong value. Accordingly, a similar process can be followed for the rest of metrics (economic performance, profitability, financial structure and short/long term solvency) shown in Table 4.1. Next, we describe how MapReduce tasks are executed to deal with each entry.

4.3.3 *Processing the SABI data panel with MapReduce*

Once the files are loaded into HDFS, the MapReduce computation process starts. MapReduce defines two entities on top of the HDFS: the JobTracker—devoted to monitor and coordinate the whole computation process—and the TaskTracker—devoted to conduct the processing operations. Hence, the JobTracker manages as many TaskTrackers as necessary to execute the data processing instances. The underlying idea behind MapReduce indeed, is to exploit data locality at the HDFS cluster and, thus, minimize the amount of data exchanged between nodes.

The MapReduce process is based on two functions that are executed sequentially: map and reduce [Dean and Ghemawat, 2010; White, 2011]. Despite its sequential execution,



```
Repsol Petróleo SA 2006 ...\t
0 \t 2242904 \t ...  \t 48258
\t 3452272 \t ...  \t 5743434
\t ...

Endesa Distribución Eléctrica
2004 ...\t 0 \t  10310213 \t
... \t  83914 \t 1612928 \t
... \t 12007055 \t ...

Ford España SL 2000 ...\t 0
\t 1373132 \t ...  \t 850 \t
1754528 \t ... \t 3460039 \t
...

Cemex España SL 2003 ...\t 0
\t 9878450\t ...  \t 48800 \t
2931482 \t ... \t 14985291 \t
...
```

```
[ Repsol Petróleo SA 2006, 5743434 \t 0 \t 2242904
\t 48258 \t 3452272 ]
[ Endesa Distribución Eléctrica 2004, 12007055 \t
0 \t 10310213 \t 83914 \t 1612928 ]
[ Ford España SL 2000, 3460039 \t 0 \t 1373132 \t
850 \t 1754528 ]
[ Cemex España SL 2003, 14985291 \t 0 \t 9878450
\t 48800 \t 2931482 ]
```

```
[Repsol Petróleo SA 2006, 1]
[Endesa Distrib.  Eléc.  2004, 1]
[Ford España SL 2000, 0]
[Cemex España SL 2003, 0]
```

Figure 4.2: An execution overview of the SABI panel data.

several map and/or reduce instances can run over different data at the same time (i.e., in parallel).

As shown in Figure 4.2, the map function, written by the user, selects the needed fields to compute the metrics shown in Table 4.1 from a given company and passes them to the reduce function with an intermediate key_i. In this way, in addition to run the map function on different rows at the same time, several map functions can process different fields of the same row.

Once map functions are finished at all nodes, the MapReduce framework merges all the values with same key and sends them to the TaskTrackers that implement the reduce function.

The reduce function, also written by the user, accepts this key_i and the set of merged values for that key (e.g., $\{value_{i1}, value_{i2}, \ldots, value_{ij}, \ldots, value_{ip}\}$) and computes the desired metric. For the sake of this experiment, the reduce output is 1 if the company passes the check or 0 otherwise. The following summarized code snippet shows the implemented map and reduce functions written in J2SE:

```
static class myMapper extends Mapper
     <LongWritable, Text, Text, Text > {

  public void map (LongWritable key,
       Text value, Context context){
           String line = value.toString();
           Pattern p = Pattern.compile("\t");
           String[] items = p.split(line);
           String[] fields = getFields(items);
           context.write(fields);
  }
}


static class myReducer extends Reducer <Text,
     Text, Text, Text>{

  public void reduce(Text key, Iterable <Text>
       values, Context context){
           context.write(key, new Text(
                checkFields(values)));
  }
}
```

The whole process is exemplified in what follows. First, each line of the SABI file is preprocessed with the scripting tool, which arranges every company entry as:

"`Company[ ]year [\t] ...[\t] shareholder contribution receivable [\t] fixed assets [\t] multi-year expenses [\t] current assets [\t] ... [\t] total assets [\t]...`",

for instance:

"`Firm1 2006 [\t] ... [\t] 0[\t] 2.242.904 [\t] ... [\t] 48.258 [\t] 3.452.272 [\t] ... [\t] 5.743.434 [\t] ...`".

The Hadoop JobTracker assigns to existing TaskTrackers the different blocks in which the files are split to do their map tasks. The defined map task returns for each company its specified four accounts per year (note that all of these accounts are separated using "\t" too). The result of the map task for the example above will be:

"⟨`Firm1 2006, 5.743.434 [\t] 0 [\t] 2.242.904 [\t] 48.258 [\t] 3.452.272`⟩",

recall that the map task (1) checks if there are empty fields in a given line—if so, it will try to either fill them with the formulae depicted at Table 4.1 or discard the whole row and, thus, will not send it to the reduce task—and (2) removes the unnecessary fields (marked with "`...`" in the example above). Then, the reduce tasks will be issued

	Agriculture		Industry		Energies		Construction & Dwellings		Services	
Year	α	β	α	β	α	β	α	β	α	β
2001	1%	39%	25%	42%	1%	56%	17%	41%	55%	42%
2002	1%	38%	24%	45%	1%	57%	18%	41%	55%	44%
2003	1%	42%	24%	44%	1%	56%	18%	43%	55%	43%
2004	1%	41%	23%	43%	1%	55%	19%	41%	55%	43%
2005	1%	43%	23%	42%	1%	56%	19%	40%	55%	43%
2006	1%	40%	23%	43%	1%	55%	19%	40%	55%	43%
2007	1%	43%	23%	44%	1%	56%	18%	40%	56%	42%
2008	1%	36%	26%	40%	2%	43%	16%	32%	54%	43%
MEAN	1%	40%	24%	43%	1%	54%	18%	40%	55%	43%

Table 4.2: Verification results (α for rows with non-empty values, β for passed).

obtaining that:

`total assets = 5.743.434, shareholder contribution receivable = 0, fixed assets = 2.242.904, multi-year expenses = 48.258, current assets = 3.452.272.`

As in this case it is satisfied, it will return the tuple

$\langle$`Firm1 2006, 1`$\rangle$.

Thus, at the end of the reduce task a file composed of the following tuples (formatted as text lines with "\t" as the field separator for each tuple) is obtained:

$\{\langle$`Firm1, 1`$\rangle, \ldots, \langle$`Firm2, 1`$\rangle, \ldots, \langle$`Firm3, 0`$\rangle, \ldots, \langle$`Firm4, 1`$\rangle\}$.

In order to extract some knowledge from firms that have mismatching data, the total amount of entries in the SABI repository from 2001 until 2008 (2.131.336 firms distributed in 266.417 entries per year) has been classified according to their working sector: agriculture, industry, energies, construction and dwellings, and services.

From the output generated by the MapReduce task, Table 4.2 is built. Each working sector has two columns: (1) α shows the ratio of firms that had non-empty values at all the required fields to compute the metrics shown in Table 4.1 from the total amount of entries, (2) β shows the ratio of these firms that passed all the verifications (e.g., at 2001, the 25% of industry firms, i.e., $266.417 * 0,25 = 66.604$ firms, had no missing fields and, from these, only the 42%, i.e., $66.604 * 0,42 = 27.974$ firms, passed the six verifications described in Table 4.1).

This section has described the implementation of the map and reduce tasks to efficiently go through the whole data repository and remove the mismatching entries. Nevertheless, it might be also interesting to compare the performance—in terms of computation time and development effort—of our method with respect to other tools designed to mine data such as project R, a free software for statistical computing, or Matlab, a high level computing language. The key lessons learnt upon this experiment are summarized in what follows.

4.3.4 *Key lessons learnt*

Data-driven applications are becoming more popular nowadays and the requirements needed to manage them are very stringent; huge volumes of data do not fit well in traditional DBMSs (see Chapter 3). Cloud storage repositories provide the proper tools and infrastructure to manage data in a scalable and efficient way as long as some properties from distributed databases can be relaxed (e.g., moving strong consistency to eventual consistency, move relational data models to key-value pairs).

This experiment explores the usage of HDFS and MapReduce from the Apache Hadoop ecosystem to efficiently (1) process financial data and (2) detect and correct errors from large data repositories related to the financial field. Specifically, a method to deal, not just storing but also computing, with the SABI data repository has been shown.

We have observed that the out-of-box experience of the HDFS—bear in mind that Apache Hadoop 0.20 has been used for the sake of this experiment—is far from the easiness featured by existing DBMS such as PostgresSQL. Several configuration parameters have to be carefully written in eXtensible Markup Language (XML) files, which requires a deep understanding of HDFS and a long time engineering effort on setting up the whole Hadoop cluster (we are aware that this issue has been improved on latest versions of Apache Hadoop [White, 2011]). Also, error logs are sometimes confusing and hard to debug due to the huge amount of events that are continuously tracked.

From the normal operating point of view, we have found that HDFS/HBase simply do what they promise: store and retrieve data. However, we have obtained disappointing results when analyzing the fault tolerance and scalability features. Specifically, we have found that it is not possible to include new nodes at run time and that data stored at failed nodes is sometimes unavailable forever (again, we are aware that this concern has been successfully addressed in newer versions of Hadoop).

From the developer point of view, we have found that it is extremely easy to program MapReduce tasks. However, there is still a long way until this distributed computing paradigm becomes familiar to naive practitioners due to the difficulties of decomposing the problem in operations of map and reduce.

From the performance point of view, we have observed that the computing time of MapReduce is very sensitive to the configured (1) number of map and reduce tasks, (2) data chunk size of the HDFS, and (3) amount of data processed by a single map task (i.e., processing individual cells versus processing entire rows). Also, we have perceived very little performance degradation when increasing the data volume from 9.4 GB to 940 GB—the maximum affordable in our cluster—probably due to the fact that nodes are underusing their computation facilities.

Overall, cloud storage repositories are—despite the aforementioned issues mostly arisen due to their immaturity—an appealing alternative to address the massive amount of data generated by nowadays applications. The architectures used to build cloud storage repositories (see Section 4.2.2) allow themselves being highly scalable, elastic, and benefiting from data locality, which fosters a new resource exploitation model referred to as cloud computing. Typically, these approaches are well-suited for OLAP applications in which consistency or response time are not a critical concern. However, there are still several OLTP applications [Cao et al., 2011] that have to deal with Big Data (i.e., demand high scalability) but either (1) cannot give up the features provided by classic distributed databases—specially in terms of consistency— or (2) are unable to fit in the existing general-purpose solutions (see Section 4.2.3). The following chapter proposes a first attempt to address this concern by combining cloud storage repositories and distributed databases.

Contribution.

1. Revision of the cloud storage fundamentals and the basic architecture layout to address Big Data.
2. Survey of the main commercial and non-commercial solutions to support cloud storage: Apache Hadoop, ElasTraS, Amazon Dynamo, Yahoo! PNUTS, and Google Spanner.
3. Hands-on experiment involving HDFS, MapReduce, and HBase from the Apache Hadoop framework to face a real-world use-case scenario.
4. Enumeration of the most relevant limitations of cloud storage repositories.

5 A CUSTOM APPROACH TO LARGE-SCALE KEY-VALUE STORES

Summary. Electric power networks, that were designed following a radial scheme on the past, are currently being rebuilt and decentralized to support renewable energies and new trends on the energy business market. This new electric distribution infrastructure model—coined as Smart Grid—demands ambitious large-scale distributed data storage facilities that traditional distributed databases are unable to offer. Current general purpose cloud storage approaches are unable to fit in the electric conservative market due to its stringent requirements. This chapter proposes a custom distributed architecture inspired by cloud storage and replication techniques used in classic databases to address the challenges posed by Smart Grids[4].

"I tried a dozen different modifications that were rejected. But they all served as a path to the final design"
— Mikhail Kalashnikov, 1947.

5.1 INTRODUCTION

Electric power networks are demanded to be highly reliable and available because they have to supply all the infrastructures of a country with exceptional service interruption. This prevents power companies from continuously updating and improving their systems because most of the changes may seriously affect critical services that they are currently providing (i.e., novel devices might not be as tested as older ones). This, together with the growth of renewable energies, leads to conservative and inefficient—due to their centralized nature—electric distribution (and generation) infrastructures that are expensive to maintain and scale.

Nowadays, the rising stringent requirements on power electric networks have driven practitioners to envisage a new way to conceive the electricity supplying and consuming models, which is known as the Smart Grid [Brown, 2008].

Classic power electric infrastructures were built following a layered and centralized scheme, where electric flows were clearly defined and advanced functionalities such as self-healing, monitoring real-time consumption, or adaptive rates were not feasible. Therefore, Smart Grids suppose a revolutionary change of the electric networks architecture—from a centralized to a distributed paradigm—that involves energy suppliers (or producers), consumers, and prosumers (i.e., producers and consumers). Overall, the goal of Smart Grids is to take advantage of current ICTs to handle the ever-increasing number of Intelligent Electronic Devices (IEDs) (e.g., circuit breakers, voltage regulators,

4 The work reported in this chapter was published as the paper entitled "An adaptive and scalable replication protocol on power Smart Grids" in the Scalable Computing: Practice and Experience (SCPE) journal, Vol 12, No. 3, 2011.

etc.) spread all over the power electric network, which should also allow companies to efficiently tune and route energy whenever and wherever it is needed.

During the last decade, several projects have been proposed to evolve traditional electric networks towards the Smart Grid concept. For instance, the gridSmart project [AEP Ohio, 2011] proposes an upgrade of the Ohio electric grid by using digital communications and automated functioning. This permits customers showing how Smart Grid technologies provide customers with greater energy control. It can also improve electricity delivery and cut energy consumption to delay the need to build more power plants. Also, the Masdar Eco city [Masdar, 2011] project proposes to build an energetically sustainable city in Abu Dhabi. Similarly, IBM and Malta's government are enrolled in the project mThink [2011] that aims at transforming the distribution network to improve operational efficiency and customer service. Specifically, they aim to replace old electricity meters with smart devices connected to an information system in order to enable remote reading, management, and monitoring throughout the entire power distribution network. However, all these approaches have mainly addressed the Smart Grid from the electric domain and, thus, have not exploited the whole benefits of ICTs. Indeed, the Smart Grid design, covers several disciplines: (1) electricity, since there are multiple power sources using different technologies; (2) networking, since all data generated by IEDs must be routed through a secure communications network; and (3) computer engineering, since these data must be properly stored and computed.

This chapter focuses on the computer engineering domain of the Smart Grid and proposes a storage architecture able to conduct data storage and ease computation tasks by combining cloud storage (see Chapter 4) and replication protocols from classic distributed databases (see Chapter 2). Specifically, this approach is slightly different than the ones used on web services [Paz et al., 2010b] or on generic cloud storage repositories [White, 2011; Palankar et al., 2008] since Smart Grids demand a set of requirements that have not been explored so far. Finally, to formally validate the proposed distributed storage protocol in absence of failures, its correctness verification is sketched.

5.2 SMART GRIDS STORAGE REQUIREMENTS

Smart Grids have become data-driven applications (see Chapter 4) from the computer engineering point of view. As opposite to classic power networks, Smart Grids own an upper management layer that takes decisions based on the vast amount of information collected by smart meters and IEDs—up to 4 TB a month [IBM-Software, 2012; INTEGRIS, 2011]—concerning the electric network status. This fact leads system designers to redefine the whole power network architecture, including its requirements and specifications, as now there is a need for storing and processing a huge amount of data besides supplying electric power. As a result, data are intensively exploited by what has been coined as smart functions (or applications) that run over the Smart Grid such as power flow monitoring, under/over voltage monitoring, load shedding, asset monitoring, or fault analysis [INTEGRIS, 2011; Gungor et al., 2013].

Despite data plays a crucial role on the Smart Grid, little effort has been made on defining the storage requirements and architecture to support all these smart functions [Kostic et al., 2005]. Therefore, a list of what a storage architecture for the Smart Grid should have—based on the experiences and lessons learnt during the INTelligent

Electrical GRId Sensor Communications (INTEGRIS) project [INTEGRIS, 2011]—is proposed in what follows:

- **Simplicity.** The conservative market of power delivery claims for solutions easy to test, maintain, and scale. Therefore, existing complex solutions with appealing features that are continuously patched and upgraded by developers, cannot fit on the Smart Grid domain.
- **Elastic scalability.** Smart Grids have an intrinsic dynamic behavior (e.g., a solar panel may stop supplying energy or an end-user consumer may switch from its local power generation device to the producer generation network). Therefore, they are required to tolerate several devices connection and disconnection with reasonable performance degradation.
- **Low network overhead.** Standard distributed data storage systems are known to add a considerable overhead to communication networks [Gray et al., 1996; Wiesmann and Schiper, 2005]. Nowadays, this issue is not usually addressed due to the ever rising capacity of network links [Corbett et al., 2012]. However, the Smart Grid relies on a heterogeneous communication network with very limited bandwidth at some points that cannot afford large traffic volumes [Zaballos et al., 2011]. Therefore, the distributed storage system has to avoid bottlenecks and ensure effective data flows.
- **Reliability and availability.** As Smart Grids are considered a critical infrastructure from any country, they are required to be highly reliable and available (some services may require to be available 99.999999% of time [Pothamsetty and Malik, 2009; INTEGRIS, 2011], which equals to 31.5 seconds of downtime per year).
- **Heterogeneous data handling.** Existing standards concerning data in Smart Grids [Kostic et al., 2005] do not currently define the format and type of information collected by IEDs. This prevents practitioners from defining a relational data model—that would probably limit the system scalability—to store and manage all these non-normalized and heterogeneous data. Therefore, data retrieved from IEDs has to be stored as key-value pairs [DeCandia et al., 2007; Chang et al., 2008], which eases the system development as well.
- **Variable data freshness.** The disperse nature of the smart functions [Gungor et al., 2013] suggests that some smart functions might tolerate working with different versions (also referred to as freshness) of the same datum. For instance, data needed to perform a load shedding (i.e., critical operations are performed upon the read values of the electric network) should be the freshest possible. On the contrary, the smart function devoted to conduct asset monitoring [INTEGRIS, 2011] might tolerate working with stale data, which would alleviate the load of the nodes that own the newest values. Indeed, tolerating different degrees of data freshness relaxes the consistency requirements of the storage system, which could contribute on further boosting its scalability.
- **Fault tolerance and recovery.** Despite the stringent requirements concerning availability and reliability, devices deployed on the Smart Grid may fail. Therefore, the storage system has to be network partition tolerant and have the capability

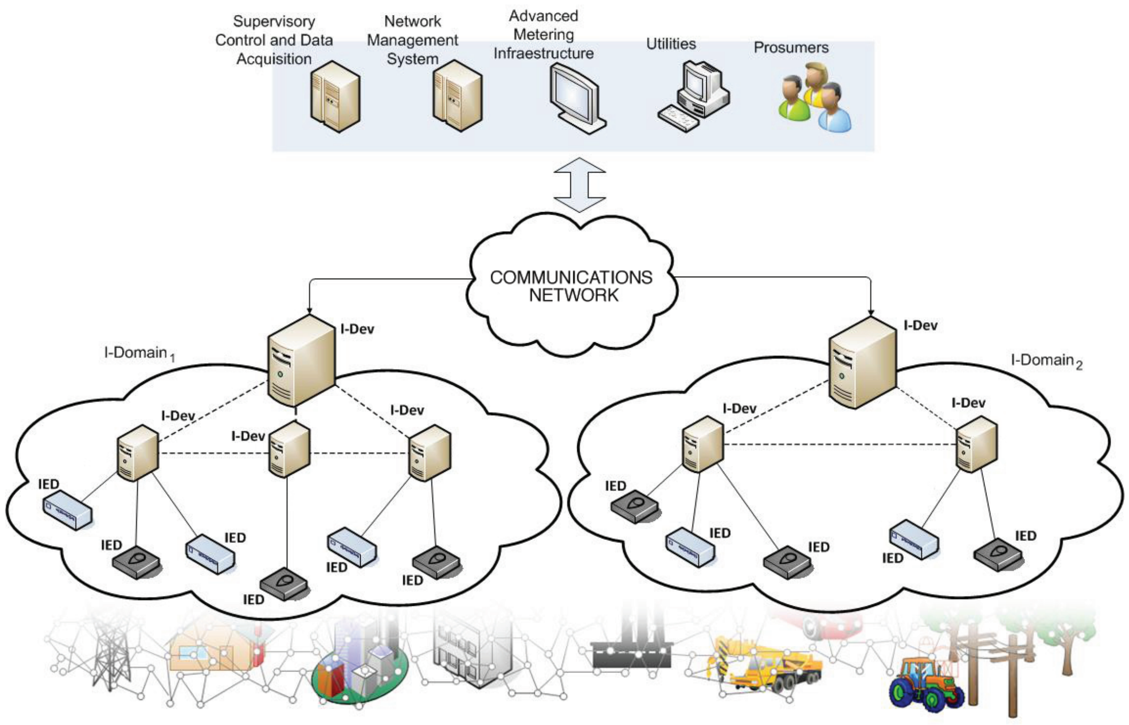

Supervisory Control and Data Acquisition
Network Management System
Advanced Metering Infraestructure
Utilities
Prosumers
COMMUNICATIONS NETWORK
I-Domain 1
I-Domain 2
I-Dev
IED

Figure 5.1: Smart Grid logical layout.

of self-reconfiguring its internal characteristics in order to keep supplying and storing data in case of failures.

Distributed storage systems—known to provide most of these requirements on many scenarios [Gray et al., 1996; DeCandia et al., 2007]—are an interesting approach to address these concerns. There are two types of distributed systems: static and dynamic. Static ones [Gray et al., 1996; Patiño-Martínez et al., 2005] require to know the identity of all nodes a priori in order to be able to distribute storage and computation and, thus, are hard to scale. (see Chapter 3). On the contrary, dynamic (also referred to as elastic) distributed systems [DeCandia et al., 2007; Palankar et al., 2008; Aguilera et al., 2009; Lakshman and Malik, 2010; Das et al., 2010b; White, 2011] that are known to be highly scalable, do not make any assumption about the system composition, which allows processes joining and leaving at will at the price of reducing some of the properties featured by static systems (see Chapter 4).

Data storage in Smart Grids demands a trade-off between both scenarios because they behave as an elastic system (i.e., devices constantly joining and leaving the system) and demand high scalability (i.e., Smart Grids are composed of thousands of IEDs), but they still need those properties that these systems tend to relax (see Chapter 4), specially consistency. The following section shows how we have taken advantage of both distributed systems strategies to propose a storage architecture for the Smart Grid.

5.3 A DISTRIBUTED STORAGE ARCHITECTURE FOR THE SMART GRID

As shown in Figure 5.1, the Smart Grid can be split into three layers: where applications are placed on top, ICTs on the middle, and power distribution is on the bottom layer.

Also, applications are linked ICTs through a communications network [Zaballos et al., 2011], and ICTs are linked with power distribution through the aforementioned IEDs.

To further favor scalability, the ICT layer has been organized in a hierarchical layout too. First, IEDs are grouped in small geographical islands referred to as I-Domains or clusters. At the same time, a set of reliable computers with limited storage and computing capabilities, coined as INTEGRIS Devices (I-Devs), have been deployed inside each I-Domain to collect the information retrieved by IEDs. Finally an I-Dev of each cluster is chosen as a leader to exchange information with other I-Domains or the applications layer.

As a single I-Dev is unable to handle all data required to solve any smart function, a distributed storage architecture is crucial to overcome the limitations of individual I-Devs.

A straight-forward way to make a distributed storage system available, reliable, and fault tolerant is by means of replication [Pedone et al., 2000]. To overcome the typical scalability issues of replication [Gray et al., 1996], we propose a novel replication protocol [Wiesmann and Schiper, 2005] inspired by primary copy (see Chapter 2) and combined with epidemic propagation, which aims to overcome the scalability limitations of primary copy and provide variable consistency. Also, we take cloud repositories (see Chapter 4) as a reference model for its architectural design [White, 2011] to promote scalability and elasticity. Overall, the proposed system architecture has (1) a metadata manager (also referred to as computation unit) that forwards user requests, and (2) a set of I-Devs that behave as storage facilities that run their own epidemic replication protocol (i.e., do not depend on the metadata manager to replicate data as cloud systems typically do [Ghemawat et al., 2003]).

5.3.1 *Epidemic propagation*

Although the number of IEDs may substantially increase as time goes by, the number of I-Devs that control them should not grow in the same way. Therefore, the proposed storage architecture focuses on the I-Devs instead of IEDs, which allows to locally tolerate the dynamism associated to IEDs. Nonetheless, the proposed system must still be robust against possible I-Dev failures, which leads us to implement some techniques commonly used in dynamic systems [DeCandia et al., 2007; Das et al., 2010b; Aguilera et al., 2009]. Therefore, we propose to base our replication strategy on epidemic propagation [Holliday et al., 2003], which consists of gradually spreading data across all I-Devs sacrificing consistency in favor of scalability and limited network flooding.

We have used the following nomenclature to tag the I-Devs when epidemically propagating data and conducting storage duties:

1. Each I-Dev in the Smart Grid has been labelled as X_{ij}, where X corresponds to the I-Dev replication role in the cluster (i.e., primary (P), pseudo-primary (PP), or secondary (S)); i is the cluster identifier, and j is the I-Dev identifier inside the cluster.

2. Each $\mathtt{cluster}_m$ has an ancestor X_{ij} that belongs to $\mathtt{cluster}_i$ ($m \neq i$).

3. Each $\mathtt{cluster}_m$ is updated through its associated pseudo-primary k (i.e., PP_{mk}) that receives updates from the $\mathtt{cluster}_m$'s ancestor.

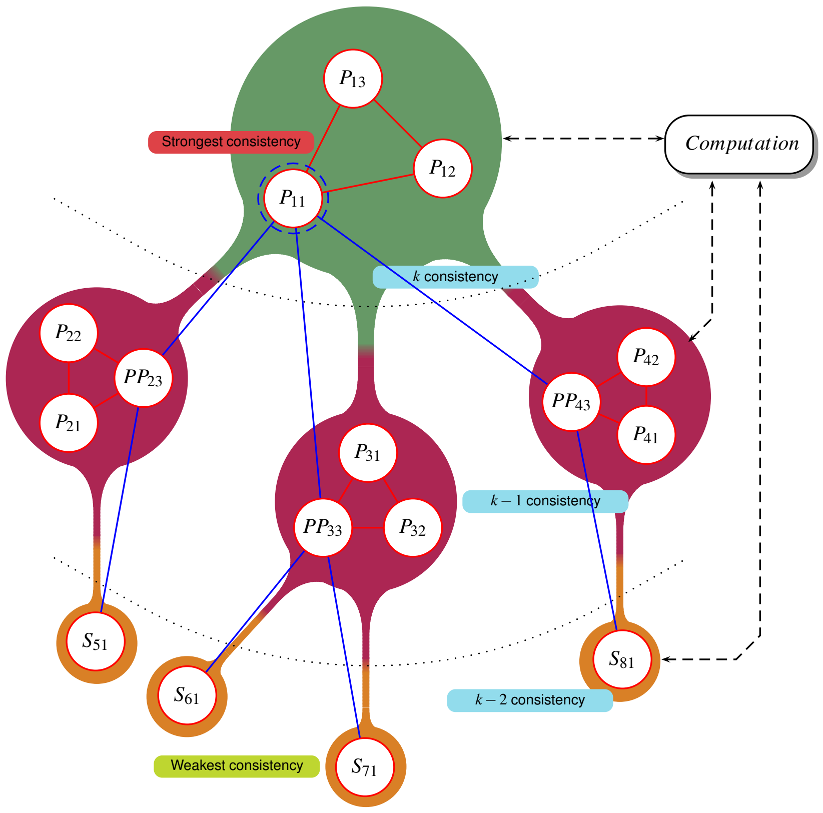


Figure 5.2: Proposed distributed storage system with a single primary node generating data.

4. Data involved in a given smart function have an associated replication depth r that quantifies the number of different clusters in which data must be replicated. If a datum is used by several smart functions with different replication depths, it will be assigned the most restrictive value (i.e., highest r).

5. The `replication chain` is a closed set of r I-Devs that exchange versions of the same data item.

For instance, Figure 5.2 shows an example with 8 clusters. Region 2 is formed by I-Devs $P_{2k} \mid k = [1,3]$, P_{21} is its primary; P_{22} is a common I-Dev; PP_{23} is the pseudo-primary; and, PP_{23}'s ancestor is P_{11}. Respectively, S_{61} is the only I-Dev on region 6 and its ancestor is PP_{33}. Also, there are four `replication chains` concerning data generated by smart meters attached to P_{11}: $\{P_{11}, PP_{23}, S_{51}\}$, $\{P_{11}, PP_{33}, S_{61}\}$, $\{P_{11}, PP_{33}, S_{71}\}$, and $\{P_{11}, PP_{43}, S_{81}\}$. Additionally, we distinguish two different types of I-Domains: (1) storage clusters that do not propagate data (regions 5, 6, 7, and 8) and (2) active clusters that do propagate data (regions 1, 2, 3, and 4).

As a result, when an I-Dev is propagating data (red lines in Figure 5.2) collected from their directly attached IEDs, it will act as a primary master (e.g., P_{11} marked with a dashed blue circle in Figure 5.2) and will treat the rest of I-Devs in its cluster as their primary slaves (e.g., P_{12}, P_{13} in Figure 5.2). When a device receives data (blue lines in Figure 5.2) from another cluster it will be acting as a repeater (pseudo-primary) (e.g., PP_{23}, PP_{33}, and PP_{43} are the pseudo-primaries of P_{11} in Figure 5.2). Finally, if an I-Dev receives updates from other clusters but it does not propagate them, it will be acting as a secondary (e.g., $S_{51}, S_{61}, S_{71}, S_{81}$ in Figure 5.2).

```
Definitions:
  1. $i \triangleq$ Current cluster ID
  2. $j \triangleq$ Current device ID
  3. $d \triangleq$ Smart meter ID
  4. $c \triangleq$ Required consistency level
  5. $r \triangleq$ Replication depth
I. Upon $Smart\_meter_{ij}(d)$ generates $data_{ij}(d)$
  1. $broadcast(cluster_i,\ j,\ data_{ij}(d),\ d)$
II. Broadcast delivery $(k,\ data_{kl}(d),\ d)$
  1. $store\_data\ (data_{kl}(d), k)$
  2. if $l = i$ then
     ⋆ $r := GetRD(data_{kl}(d),\ d)$
     ⋆ if $r > 0$ then
        ◊ $list\ :=\ <i, j>$
        ◊ $multicast(neighbors_{ij},\ list,\ data_{kl}(d),\ r-1)$
III. Multicast delivery $(list,\ data_{kl}(d),\ r)$
  1. $store\_data\ (data_{kl}(d), last\_item(list))$
  2. if $r > 0$ then
     ⋆ $destination\ :=\ (neighbors_{ij} \cap list) \setminus$
           $(neighbors_{ij} \cup list)$
     ⋆ $list\ :=\ list\ \cap\ <i, j>$
     ⋆ $multicast(destination,\ list,\ data_{kl}(d),\ r-1)$
IV. Data request $(data_{kl}(d),\ c)$ from $source$
  1. if $\nexists\ data_{kl}(d)$ then
     ⋆ $unicast(source,\ nil,\ -1)$
  2. else if $c\ \geq\ GetConsistency(data_{kl}(d))$ then
     ⋆ $unicast(ancestor(data_{kl}(d)),\ data_{kl}(d),\ c)$
  3. else
     ⋆ $unicast(source,\ data_{kl}(d),\ c)$
```

Figure 5.3: Formal specification of the replication protocol at IED_{ij}

Then, once the primary master has sent its data to a pseudo-primary node of another cluster, a recursive process starts where each pseudo-primary looks for another pseudo-primary in another neighboring cluster to propagate its data. This process finishes when either (1) there are no more clusters, or (2) there is a cluster that has no more neighbors (i.e., S_{51}, S_{61}, S_{71}, and S_{81}) in case of full replication, or (3) when data has been replicated to all I-Devs of the replication chain.

It is worth mentioning that any I-Dev belonging to an active cluster may simultaneously adopt different roles (i.e., primary master, primary slave, pseudo-primary, and secondary) at the same time according to the replication protocol configuration. For the sake of simplicity, from now on it is assumed that there is a single primary master I-Dev generating data on the whole Smart Grid as depicted in Figure 5.2.

This hierarchical layout is flexible enough and able to satisfy the data freshness requirements [Plattner et al., 2006; Armendáriz-Iñigo et al., 2007] imposed by smart functions [Chuang and McGranaghan, 2008]. Specifically, each time a smart function needs to calculate the result of a given smart function, it first attempts to use data from its nearest neighboring cluster. If data contained on that cluster has a consistency level k greater than l—where l is the freshness level required by the function—it will use that cluster to perform computation. Otherwise, it will get redirected to its ancestor node with a freshness index $k-1$ and repeat the operation.

This way of propagating data allows the system to (1) balance data requests between different clusters according to links status and congestion, (2) find the most appropriate datum version for computing a smart function, and (3) circumvent the traditional problems of scalability and availability typically found in these approaches [Patiño-Martínez et al., 2005; Armendáriz-Iñigo et al., 2007; Jiménez-Peris et al., 2002; Wiesmann and Schiper, 2005].

Hence, the replication protocol has to decide when and where data are propagated [Wiesmann et al., 2000]. Such decisions must be taken to avoid propagation loops and according to the smart function requirements (e.g., flow monitoring will require faster updates than asset management).

5.3.2 *Proposed replication protocol*

Epidemically propagating data across the hierarchical layout depicted in Figure 5.2 ensures that update operations are applied sequentially, which is a good basis for providing eventual consistency among I-Devs from different clusters (i.e., if no more data are generated all clusters will converge to the same state). Additionally, it is possible to maintain strong consistency inside each cluster—without compromising scalability—because the number of I-Devs inside a cluster is reasonably small (i.e., up to ten).

As a result, we have obtained a hybrid replication protocol whose formal specification is shown in Figure 5.3 and described in what follows:

1. When an I-Dev (i.e., primary master, P_{11} in Figure 5.2) receives data from its IEDs it eagerly replicates them by $\mathtt{broadcasting}$ (step I in Figure 5.3) these data to all devices within its cluster (i.e., primary slaves P_{12} and P_{13} in Figure 5.2) using active replication (step II.1 in Figure 5.3).

2. If the replication depth r associated to these data is greater than 0, the primary master lazily and passively replicates them to other clusters avoiding replication loops and $\mathtt{multicasting}$ relevant meta-data.

   In order to avoid replication loops (i.e., different I-Devs from the same cluster holding a different number of versions from the same datum), each I-Dev must build a $\mathtt{list}$ with the ancestors of data it is currently processing and remove all I-Devs of its neighbor list that are in the ancestor list. Recall that each primary or pseudo-primary I-Dev has assigned only one pseudo-primary per region at a time.

   Regarding the neighbor (i.e., pseudo-primary) discovery, we assume that given a neighboring cluster B from cluster A, all the nodes in cluster A will deterministically choose the same pseudo-primary from cluster B. If this pseudo-primary fails or there is a high network congestion, the next pseudo-primary chosen will be the one with the lowest identifier in cluster B. For example, in Figure 5.2, if P_{33} fails, P_{32} and P_{33} will also chose S_{61} and S_{71} as their pseudo-primaries.

   Once the neighbor list has been pruned (and updated), the I-Dev $\mathtt{multicasts}$ (1) the ancestors $\mathtt{list}$, (2) the stored $\mathtt{data}_{kl}(d)$ and (3) a decremented value of replication depth to its neighbors (i.e., PP_{23}, PP_{33} and PP_{43} in Figure 5.2) as shown in step II.2 in Figure 5.3.

3. This replication process is repeated while the replication depth is greater than 0 as shown in step III in Figure 5.3.

   The first time data from $\mathtt{smart\ meter}_d$ achieves the latest pseudo-primary (r has reached the 0 value), or secondary (i.e., S_{51},S_{61},S_{71},S_{81} in Figure 5.2) this device will send to the metadata manager its identifier.

   This ensures that the metadata manager will know where to find the stalest and the newest replica of data associated to IED_d.

4. When an I-Dev receives a query, it first checks whether the freshness level k of its stored data is enough to perform the computation. If it is greater than the

one required and the system can afford more load, it will give back its stored data, otherwise it will forward this query to its ancestor as shown at step IV in Figure 5.3.

Overall, this approach combines the following replication strategies typically used in classic distributed databases (see Chapter 2):

- **Inside a cluster.** Active replication is used to ease fault tolerance. Eager replication is used to keep strong consistency. Update-everywhere replication is used to dynamically perform load balancing.
- **Between clusters.** Passive replication is used to minimize the size of the network messages. Lazy replication is used to alleviate scalability issues. Primary backup is used to avoid expensive—in terms of network messages—synchronizations between I-Devs from several clusters.

The obtained features regarding data consistency and fault tolerance derived from using this replication approach are discussed in the following section.

5.4 SYSTEM DISCUSSION

So far, we have shown that our proposed distributed storage system (1) is reasonable simple (i.e., see Figure 5.3), (2) is scalable and elastic (i.e., I-Devs and IEDs can be added or removed at will with little impact thanks to the epidemic propagation running on a hierarchical layout), (3) adds low network overhead (i.e., passive and lazy replication has been used among clusters), and (4) it promotes high availability and reliability (i.e., thanks to replication [Pedone et al., 2000]). Nonetheless, there are still some important topics that are worth to further discuss: data freshness, performance improvements, fault tolerance, and another possible domain of application.

5.4.1 *Data freshness and consistency*

The interval of time in which smart functions collect information from I-Devs (also referred to as data periodicity) plays an important role on establishing the freshness of data stored at each replication cluster. For instance, voltage measurements might be sampled with a higher frequency than heat measurements, which means that several freshness degrees from different data can cohabit in the same cluster. Hence, each datum is stored together with the timestamp k in which it was originally acquired—as similarly done in stream warehouses with the version technique used [Botan et al., 2010; Golab and Johnson, 2011].

Indeed, there exists a close relationship between the data freshness k and consistency—understood as the property which states that all the members in the replication chain own the same version of the datum with timestamp k—: data with high values of k correspond to the latest measurements but are potentially weak consistent. Analogously, data with low values of k correspond to the oldest measurements but are potentially strong consistent (i.e., that version of the datum has been seen by all I-Devs of the replication chain).

Overall, our proposed approach provides eventual consistency among clusters and strong consistency inside a cluster, which reduces the availability of fresh data in favor of scalability. The rationale behind this design decision remains on the fact that the most computing intensive (i.e., the ones that need data replicated at different sites to perform parallel computations) smart functions [INTEGRIS, 2011; Gungor et al., 2013] can afford dealing with stale—but consistent—data.

5.4.2 *Performance improvements*

Although our system is designed according the cloud repositories fundamentals (see Chapter 4), its scalability is bounded by the proposed data replication protocol, which is inspired by those ones used in (poorly scalable) distributed databases. Therefore, at some point in the (near) future it might be necessary to conduct some configuration adjustments in order to minimize the performance degradation as the system grows.

Master cluster reduction. So far, we have assumed that all I-Devs of a given cluster participate in the active replication of all data, which is not necessary at all: although the number of I-Devs belonging to a cluster can be in the range of tens, the replication process could be speeded up by selecting a reduced set of representatives. It is well known that active replication does not scale well [Gray et al., 1996; Wiesmann and Schiper, 2005] and with the proper selection of representatives the rest can become secondaries of each representative (inside the same cluster).

Dynamic replication depth tuning. Dynamically adapting the value of replication depth for each datum according to the availability demands of every smart function [INTEGRIS, 2011] might improve the usage of I-Devs' storage resources. In fact, we have not specifically stated how the replication depth is set, neither augmented or decreased. It can be set by the system administrator but it could also be dynamically adjusted as a function of the demands coming from the metadata manager. Moreover, it could be configured autonomously in case of disaster upon a rapid evaluation of certain functions (e.g., accounting). Therefore, it might be interesting to incorporate a learning classifier system (e.g., eXtended Classifier System (XCS) or sUpervised Classifier System (UCS)) [Wilson, 1995] and build a cognitive system [Holland, 1992] in order to (1) evaluate the whole system status, (2) predict the optimal value of the replication depth for each data item, and (3) adapt the system itself to the Smart Grid dynamism.

Distributed computing. Our proposed architecture allows to perform distributed computation on the read operations. Specifically, it is especially suitable for those distributed computing tasks that might benefit from data aggregation. Due to the fact that required data travel across the replication chain (depending on the required consistency level k) each I-Dev can perform a piece of the final computation required. For instance, to sum all the voltage measurements collected by a given IED, the metadata manager should decide that lower I-Devs of its associated replication chain have to sum the older measurements while higher I-Devs have to sum the most recent measurements—still unavailable on most I-Devs—, which would speed up the computation process and distributed the overhead among all I-Devs of a replication chain.

To make this possible, an extra software layer should be placed at the computation unit in order to track all the distributed operations and consistently aggregate them as done by MapReduce [Dean and Ghemawat, 2010; White, 2011].

5.4.3 *Fault tolerance*

At the ICT layer, there are two major possible fault sources in the Smart Grid: the communication network and the distributed storage architecture. So far, a lot of successful research has been conducted on obtaining a reliable communications system for the Smart Grid [Pothamsetty and Malik, 2009; Galli et al., 2010; Zaballos et al., 2011; INTEGRIS, 2011], which allows us to focus our efforts on the distributed storage architecture. Hence, our current goal is to implement a policy to (1) detect node failures, (2) ensure that the global system will behave properly in case of I-Dev faults, and (3) recover failed nodes.

Faulty I-Devs detection. As in many other distributed systems, we assume that any I-Dev may fail according to the crash model [Cristian, 1991]. To detect crashed I-Devs, the metadata manager continuously exchanges small heartbeat messages with all I-Devs—as cloud storage repositories do—and uses a timeout policy to discard faulty devices from any replication chain.

If an I-Dev started behaving in an arbitrary manner (i.e., exhibiting a byzantine behavior [Kapitza et al., 2012]), it would be either because it is returning or propagating an arbitrary or older (though valid) version of a variable. We control this by adding a digest to the value stored, similar to what it has been proposed in [Castro and Liskov, 2002; Pedone et al., 2011]. As a result, whenever we find a mismatch between them we can notify the metadata manager that this I-Dev should be shut down.

System behavior under faults. Actually, we can face two different failure situations: the failure of a primary I-Dev (e.g., P_{11} in Figure 5.2) and the the failure of a pseudo-primary I-Dev (e.g., PP_{23} in Figure 5.2).

In the former, the proposed approach inherits the advantages of active replication (i.e., all I-Devs in a cluster are all the time sharing the same state as the cluster's primary) and, thus, the system is able to recover easily: when a primary master fails, any other primary-slave can immediately take over the situation. In the latter failure scenario, as soon as the ancestor$_A$—belonging to cluster A—of the failed I-Dev$_B$—belonging to cluster B—detects its unresponsiveness, it will ask the metadata manager for a list of available pseudo-primaries from cluster B. If no more pseudo-primaries (or secondaries) are available (e.g., cluster 7 in Figure 5.2), ancestor$_A$ will send a message to the metadata manager notifying that it is the new last I-Dev of the replication chain.

I-Dev recovery and role reassignment. When an I-Dev fails it is necessary to transfer its status to another I-Dev. Such state transfer depends on the I-Dev current role(s): secondary, primary, or pseudo-primary.

If the role is a secondary, the recovery process consists of transferring all the information stored at the ancestor's crashed I-Dev to another I-Dev.

If the role is a primary, it is not necessary to transfer any state information thanks to the fact that all primaries inside every cluster are eagerly replicated and, thus, fully synchronized. If the crashed primary was acting as an ancestor (e.g., P_{11} in Figure 5.2), the metadata manager will reassign that role to another I-Dev from the same cluster (e.g., P_{12} in Figure 5.2).

If the role is a pseudo-primary, the takeover process is not that straight-forward because pseudo-primaries perform passive and lazy replication. The challenge here is that the takeover solution in a pseudo-primary implies that the previous versions of a given datum might be lost. A very first approach to address this concern would consist of transferring the full state to the new successor from the ancestor (recall that it belongs to another cluster and this could be costly). However, this alternative has some drawbacks: (1) it might affect the availability of the system since transferring the whole data may affect the transmission of new data to the successors, and (2) the amount of data to be transferred might be so big that a new successor could never catch up with the current state of the system [Vilaça et al., 2009]. Alternatively, it is possible to perform a partial state transfer of data to other I-Devs of the cluster during normal operation. This implies that the replication algorithm has to ensure that each pseudo primary has a set of secondaries in its associated cluster where the updates get also propagated asynchronously. Therefore, when a given pseudo-primary fails, the amount of data to be transferred between clusters is greatly reduced. Nevertheless, this has to be tested and checked in a real-world scenario to find out the proper number of pseudo-primary slaves and the amount of data transferred.

Additionally, this replication approach enables to feature what has been recently coined as k-safety [Stonebraker and Weisberg, 2013], which is a way to trade fault tolerance with availability [Brewer, 2012] and results very convenient for elastic environments.

5.4.4 *An alternative domain of application*

Stream warehouses [Golab and Johnson, 2011], where a continuous stream of data has to be stored, have many similarities with Smart Grids. In order to overcome the classic update and query consistency issues found in these scenarios [Golab and Johnson, 2011], our architecture guarantees that there will never exist multiple attempts to write the same datum at different places because (1) every datum has been assigned a replication chain, (2) replication chains are loop-less, and (3) multi-version techniques [Botan et al., 2010] are used to maintain a notion of consistency.

In fact, when our architecture is required to store data stream measurements [Vogels, 2009; Golab and Johnson, 2011], the most recent data versions will be always located closer to their sources; whereas oldest versions might have already reached the furthest devices in the replication chain. However, if a given measurement is not so frequently taken, then the most recent version will be found anywhere in its associated replication chain.

The following subsection proposes an analytical study of the scale out factor of our protocol in order to obtain some arguments that assess the viability of our approach before deploying it in a real-world scenario.

5.5 ANALYTICAL EVALUATION

As one of the major goals of our proposal is achieving high scalability, we have used the analytical model and notation from Serrano et al. [2007] that estimates the scale out factor in a replicated database when there is an increase of the number of sites and replicas. In the following, we briefly describe the adaption to our system model and proceed with the computation of the scale out factor of our approach.

The scale out factor determines how the performance of the global system is improved or degraded by using replication. As shown in Equation 5.1, this is computed as the sum of work executed at each replica divided by the processing capacity of a non-replicated database.

$$\text{Scale Out} = \frac{1}{C}\sum_{i=1}^{n}\sum_{j=1}^{m} C \cdot U_{ij} \cdot ACC_{ij}. \tag{5.1}$$

According to Serrano et al. [2007], we have that C is the processing capacity of a non-replicated database, n is the number of I-Devs in our Smart Grid, m is the number of stored objects, U_{ij} defines the location of object j at site i, and ACC_{ij} defines the accessibility (i.e., access rate) of object j at site i. Hence, if the object j is at site i, then $U_{ij} = 1$, which defines the replication schema.

However, Equation 5.1 assumes that read and update operations launched against the replicated distributed system are uniformly spread across all replicas. Hence, this equation is not directly applicable to our proposal because our solution has an in-built load balancing mechanism that redirects operations to replicas according to the required consistency level k. Thus, we show how we have adapted the analytical description of the replicated system to fit in our system characteristics.

Recall that the term $(C \cdot U_{ij} \cdot ACC_{ij})$ is equivalent to $(C_r \cdot ACCR_{ij} + C_u \cdot ACCU_{ij})$ where C_r is the read processing capacity, C_u is the update processing capacity, $ACCR_{ij}$ is the read accessibility, and $ACCU_{ij}$ is the update accessibility. So, replacing this expression in Equation 5.1 we obtain Equation 5.2 that is now useful in our replication protocol—actually, Equation 5.1 can be considered a generalization of Equation 5.2.

$$\text{Scale Out} = \frac{1}{C}\sum_{i=1}^{n}\sum_{j=1}^{m} C_r \cdot ACCR_{ij} + C_u \cdot ACCU_{ij}. \tag{5.2}$$

We now evaluate the different replication strategies for a system from 10 to 80 sites, 10 objects, 80% of read operations and 20% of write operations to all objects. Objects and operations on them are evenly distributed. We also assume that all sites have the same processing capacity (i.e., all I-Devs have similar specifications and storage capabilities).

We have compared the scale out factor obtained in our proposed protocol against the replication strategies proposed in [Serrano et al., 2007]: (1) full replication—all sites contain the same data—and (2) partial replication —a reduced set of sites contains a given data. In the case of the latter, we have chosen to replicate each data item in $n/2$ sites in order to obtain comparative results. The scale out evaluation of these protocols is shown in Table 5.1.

Ideally, the scale out factor should be equal to the number of sites of the system, which means that all incoming updates would be processed without saturation. We can see that with a full replication scheme the scale out is quite poor: 40 with 80 sites. When the

Sites	Full Replication	Partial Replication (50%)	**Hybrid approach**
2	2	2	2
10	10	9,8	9,82
20	15,7	16,9	18,15
40	25	28,2	30,7
80	40	48,2	53

Table 5.1: Analytical scale out evaluation of the replication protocol for Smart Grids.

number of sites increases, the system cannot scale anymore as the full replication policy forces that all updates have to be sent to all replicas which produces a considerable overhead.

In contrast, with the partial replication (limited to half of the replicas) the system scales slightly better because the cost of propagating the replicas is not that high (scale out of 48,2 with 80 sites). Finally, the scale out of our epidemic replication protocol is even better (53 at 80 sites). In this case, I-Devs that have the most recent data version have a higher C_u and lower $ACCR_{ij}$ because most read operations are executed in I-Devs that do not have necessary the last version of the data. For these reasons, we show that the scale out is better than traditional full replication and partial database replication protocols.

Overall, we have improved the scale out of a traditional replicated system by balancing read and write operations and, thus, providing the system with the ability to afford more overhead associated to write operations. To further support the feasibility of our approach, the following section roughly verifies the correctness properties of this system if all nodes behave properly and faults never occur as done by Das [2011].

5.6 CORRECTNESS GUARANTEES

Distributed algorithms are said to be formally correct when their liveness and safety properties are satisfied and shown to be correct. Regarding the liveness property, it can be best seen as something good will eventually happen; while, the safety property can be stated as nothing bad will happen.

Our proposal needs to propagate the measures from a given IED from cluster to cluster up to its replication level; hence, changes need to get propagated and applied in the same order in all pseudo-primaries. Both asserts constitute the liveness and safety properties of our system.

Next, we point out some guarantees of the system extracted from the previous sections in order to have enough arguments to warrant the correctness properties.

GUARANTEE 1. Any primary (or pseudo-primary) will never send the same data item to more than one device per neighboring cluster.

This is guaranteed since data ancestors are excluded from the device's neighbor list as shown in step III.2 in Figure 5.3.

GUARANTEE 2. The metadata manager knows which is the last pseudo-primary (or secondary) of the replication chain.

As described in Section 5.3.2, when a device notices that a new data item has gone through all its replication depth ($r = 0$), it will send a message to the metadata manager identifying itself.

GUARANTEE 3. Any device belonging to a master cluster has always the latest version of data generated by any smart meter within that cluster.

This is satisfied since data is eagerly replicated to all devices of the master cluster as shown in steps I.1 and II.1 in Figure 5.3.

From this guarantees, we can state the safety and liveness properties of our system. As the system behaves different depending on whether it is replicating data (also referred to as write) or executing a query (also referred to as read), both properties (safety and liveness) must be analyzed in two facets: read and write.

5.6.1 *Safety properties*

The safety properties of our architecture are stated by the following claims:

CLAIM 1. **Safety on write.** Safety on write operations is guaranteed since there will never occur a situation where the same data is being written from two or more different sources.

This is guaranteed since (1) there is only one IED_d generating $data_d$, (2) point to point communication channels do not disorder messages, and (3) the $neighbor$ function will never find more than one device per cluster as stated in Guarantee 1.

CLAIM 2. **Safety on read.** There will always exist a consistent version of the requested data item queried by the metadata manager.

This is assured provided that the replication protocol (Figure 5.3) guarantees consistent writes throughout the whole replication chain.

5.6.2 *Liveness properties*

The liveness properties of our architecture are stated by the following claims:

CLAIM 3. **Liveness on write.** Liveness on updates is trivially assured by Guarantee 1.

Data generated on the smart meter will follow the replication chain and being consistently written at each device until the replication depth reaches a value of 0 or there are no more neighbors.

CLAIM 4. **Liveness on read.** Liveness on data reads is guaranteed provided that Claim 2 is accomplished.

The metadata manager will always send queries to the last device of the replication chain. If data contained in it have not the required consistency level k, the device will redirect the query to its ancestor. This will happen in a recursive way until the required level k is found. If any device can reach the required level k the primary will give back with its latest version which is strongly consistent as provided by Guarantee 3.

Overall, in this chapter we have presented a theoretical approach to distributed storage for Smart Grids taking advantage from many techniques used in distributed systems, which to the best of our knowledge have never been put together before. We have defined a way to distribute—based on epidemic propagation—and store information across the Smart Grid so that the computation needed for certain smart functions can be greatly reduced. We have detailed the replication protocol based on epidemic updates and checked its formal correctness. As the proposed approach satisfies the safety and liveness properties, it is reasonable to conclude that it is formally correct and its implementation might be feasible, which are important arguments to start its deployment in a real-world scenario. The following chapter describes how this approach has been deployed together with the other subsystems of the Smart Grid.

Contribution.

1. Revision of the storage requirements and challenges posed by Smart Grids.
2. Description of a key-value distributed storage architecture for Smart Grids inspired by cloud repositories.
3. Specification of a data replication protocol that combines epidemic propagation with replication techniques used in transactional databases.
4. Validation of the proposed replication protocol formal correctness.

6 A CLOUD-BASED INFRASTRUCTURE FOR POWER SMART GRIDS

Summary. Smart Grids embrace several poorly correlated research disciplines that have been rarely put together so far, which has led to a considerable number of partial and isolated solutions. This chapter presents a fully integrated ICT infrastructure for the Smart Grid that incorporates a secured communications layer, a cognitive system, and the distributed storage architecture—with its associated replication protocol—proposed on the previous chapter. Conducted experiments attest the feasibility of our solution and exhibit its benefits when facing real-world scenarios[5].

"The ideal engineer is a composite: He is not a scientist, he is not a mathematician, he is not a sociologist or a writer; but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems"
— Nathan Washington Dougherty, 1955.

6.1 introduction

Smart Grids design and implementation, as opposite of classic electric distribution networks, covers several disciplines (also referenced as subsystems) which are rarely addressed as a whole: devices in the Smart Grid have to communicate between each other, which demands some notions about Telecommunications; also, these devices deal with critical data prone to malicious attacks, which demands some notions about Security and Telematics; and finally, all these massive data have to be properly stored, which demands some notions about Data Management. Although each discipline is dependent from the others within the Smart Grid domain, practitioners tend to tackle each problem independently (i.e., assuming the best behavior from other subsystems).

This situation might lead to potentially conflicting situations where a given subsystem—with a reduced and partial view of the whole Smart Grid—pursuing its greatest throughput is forcing others to behave poorly and, thus, degrading the overall performance (e.g., the security module may block all devices what would prevent the storage subsystem making any datum available). Therefore we propose to include an entity, herein referred to as cognitive system, able to (1) evaluate the status of all subsystems—network, security, and data management (also referred to as data storage)—, (2) extract a global knowledge of the whole Smart Grid, and (3) tune it accordingly to optimize its performance.

5 The work reported in this chapter was published as the paper entitled "The information system of INTEGRIS: INTelligent Electrical GRId Sensor communications" in the IEEE Transactions on Industrial Informatics journal, Vol 9, No. 3, 2013.

The purpose of this chapter is to describe the integration and deployment of the distributed storage system—and its associated replication protocol—proposed in Chapter 5, with a secured communications layer and a cognitive system at the Smart Grid's ICT layer. Specifically, such integration covers the following blocks developed along the INTEGRIS project [INTEGRIS, 2011]:

- **Quality of Service (QoS)-aware communication module.** It enables a proper heterogeneous network management aimed at integrating different solutions for the Smart Grid communication technologies which adapt themselves to the proposed link layer topology control.
- **Security.** It provides a reliable infrastructure to ensure relevant data security in order to collaborate with the defined cognitive system and to contribute to the global optimization of the INTEGRIS system.
- **Cognitive System.** We have implemented an intelligent system that is aware of the environment in which it is embedded on to autonomously and dynamically take decisions affecting the behavior of the whole Smart Grid.
- **Distributed storage system.** We have deployed the cloud-inspired data storage repository proposed in Chapter 5 able to replicate data collected from IEDs.

The remainder of this chapter is devoted to (1) briefly review the communications, security, distributed storage, and cognitive systems infrastructure used on INTEGRIS [INTEGRIS, 2011], (2) describe the metrics and the middleware used to evaluate the whole ICT layer, and (3) show the obtained experimental results.

6.2 INTEGRIS SECURITY AND COMMUNICATIONS TECHNOLOGY

Smart Grids link many distinct types of IEDs demanding very different QoS levels over different physical media. Indeed, this kind of data communications network is not exempt from the growing needs of QoS. In addition, availability and secured communications are also crucial for the proper network operation [Carcano et al., 2011]. Communications between the control center, IEDs, and substations are carried out through a wide variety of technologies according to each segment physical characteristics (e.g., Broad Band Power Line Communications (BB-PLC) [Galli et al., 2010], Ethernet or WiMAX [INTEGRIS, 2011; Zaballos et al., 2011]). Hence, within the Smart Grid domain these technologies have to coexist and interact between each other, which drives practitioners to consider Active Network Management (ANM) techniques to coordinate the whole communication network [Yang et al., 2011].

To cope with such a highly heterogeneous environment with strict QoS and security constraints [Vallejo et al., 2012], the underlying communications infrastructure that enables Smart Grids has to be carefully addressed and designed. This section reviews the proposed trustable framework for INTEGRIS devoted properly manage all communications in the Smart Grid and provide end-to-end QoS.

6.2.1 *Data communications for the Smart Grid*

Smart Grids run novel smart functions that traditional electricity distribution networks are currently unable to offer [Gungor et al., 2013]. To achieve such a goal, Smart Grids

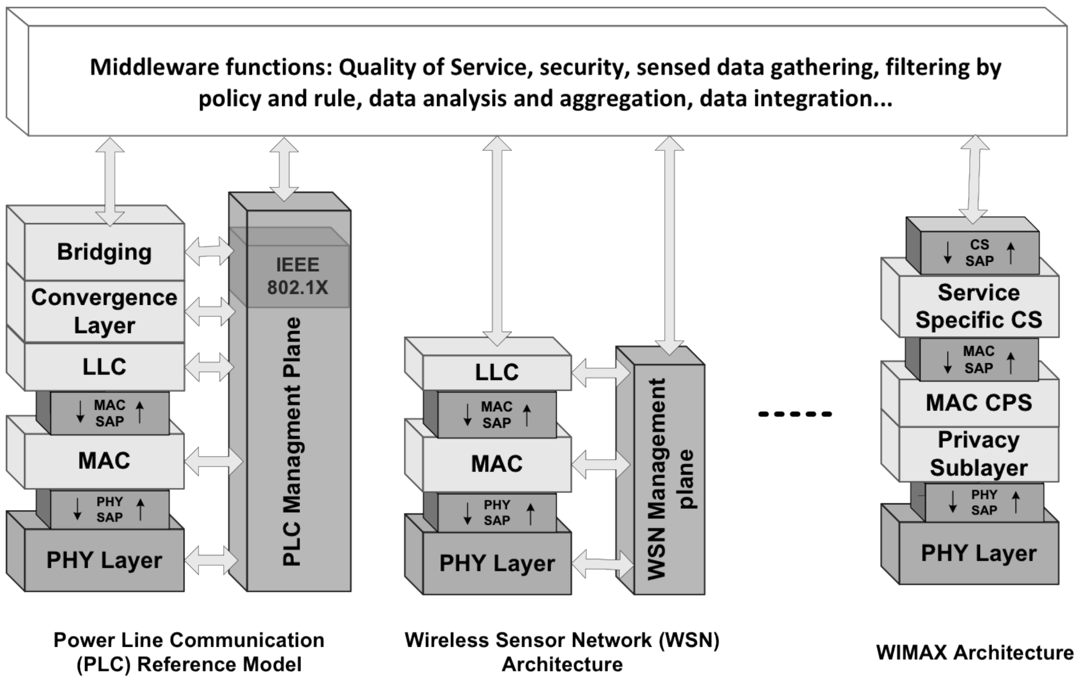

Middleware functions: Quality of Service, security, sensed data gathering, filtering by policy and rule, data analysis and aggregation, data integration...
Bridging
Convergence Layer
LLC
MAC SAP
MAC
PHY SAP
PHY Layer
IEEE 802.1X
PLC Managment Plane
LLC
MAC SAP
MAC
PHY SAP
PHY Layer
WSN Management plane
CS SAP
Service Specific CS
MAC SAP
MAC CPS
Privacy Sublayer
PHY SAP
PHY Layer
Power Line Communication (PLC) Reference Model
Wireless Sensor Network (WSN) Architecture
WIMAX Architecture

Figure 6.1: INTEGRIS middleware for the Smart Grid's network communications management.

deal with a very diverse array of ICT requirements, some of them very demanding in terms of latency or reliability levels (see Chapter 5), that are far beyond of what it is needed in other ICT-based systems [Pothamsetty and Malik, 2009].

ICTs are recognized as a key enabler of the Smart Grid [Arnold, 2011; Campos et al., 2014] and several European research projects have been launched to design an adequate data communications infrastructure to support them [Repo et al., 2011]. To address this issue, the international community, among many other efforts, is standardizing some protocols such as the IEC-61850 protocol [IEC, 2003] and defining the requirements that the communications network should meet to achieve the high degree of QoS expected from the smart functions executed on the Smart Grid [IEEE, 2004].

Unfortunately, current network technologies have several troubles in meeting these requirements. For instance, the Internet Protocol—widely used in ICT systems—convergence times are on the order of 10-30s [Eramo et al., 2008; Zaballos et al., 2010], which is far from the downtimes demanded by Smart Grids (see Chapter 5). Regarding the link layer fault tolerance, the original Spanning Tree Protocol (STP) had recovery times on the order of 30 seconds [Touch and Perlman, 2009]. Nevertheless, although newer versions of this protocol (e.g., Rapid STP) claim much better recovery times, STP intrinsically uses a single path, which makes it more vulnerable to failures and, thus, not suitable for the Smart Grid domain.

So far, alternative approaches have been proposed to address the problem of reducing the recovery times of communication networks with Parallel Redundancy Protocol and High-Availability Seamless Redundancy [IEC, 2007b]. In addition, two new protocols have been recently proposed to solve somewhat similar problems: (1) TRILL [Touch and Perlman, 2009], which proposes the concept of RBridges to obtain failover delays that favorably compare with STP and Rapid STP, and (2) IEEE 802.1aq [IEEE, 2011], which is currently being standardized by the Institute of Electrical and Electronics Engineers (IEEE).

Thus, we propose a novel and flexible network communications infrastructure that consists of blending heterogeneous OSI layer 2 technologies (e.g., BB-PLC, Wireless, Fiber Optics, etc.) by means of a custom version of TRILL and managed by a custom communications middleware as shown in Figure 6.1. Also, this communications middleware is responsible from providing a secure communication network, whose policies are reviewed in the following subsection. Recall that all these facilities have been incorporated to all I-Devs that populate the Smart Grid (see Chapter 5). Therefore, in addition to conduct storage and replication roles, I-Devs are devoted to act as Ethernet bridges that enable low latency data communications across the Smart Grid.

6.2.2 *Secured communication architecture*

Any device connected to a communication network should include security capabilities. These systems are usually scanned by hackers to exploit their vulnerabilities or used to participate in attacks towards the most valuable targets of a country. Indeed, it is envisaged that the Smart Grid will become a primary target of hackers and cyber-terrorism because it is widely considered as a critical infrastructure, which is a real concern for the industry. For the sake of the INTEGRIS project we have defined a set of security measures to make our solution deployable and operative.

Security in Smart Grids is essential for the survival and feasibility of the global electricity distribution concept [Metke and Ekl, 2010]. Smart Grids inherit all the Internet security vulnerabilities and add some others due to the different standards, applications, requirements, and actors interacting together.

In fact, our proposal aims to balance the many—and sometimes conflicting—security goals of the different actors and subsystems at the Smart Grid and, thus, accommodates a large and dynamic set of security mechanisms. It is worth mentioning that the integration of different types of communication protocols and energy sensor technologies results in a significant variation of the operational capacities at different segments of the Smart Grid. This means that different security levels [Curino et al., 2011b] must be considered for each network segment in order to prevent resource-constrained I-Devs from limiting the security guarantees in the rest of the network.

The Smart Grid has its own specificities concerning security that need to be considered indeed. According to the IEEE directives, Smart Grids security must only comply with the IEC62351-6 standard, which basically forces the use of Transport Layer Security (TLS) [IEC, 2007a]. However, TLS only covers the security at OSI layer 4, which clearly is not enough in Smart Grids that urge multilevel security.

Therefore, we propose to include the IPsec protocol at OSI layer 3. Despite the different OSI layer 2 technologies running on different sides in the Smart Grid, I-Devs must be able to manage the security policies defined for each technology. Additionally, a rigorous consideration of the security aspects at the bottom OSI layers of the Smart Grid is critical, since there may be many possibilities at this level leading to important Denial of Service vulnerabilities if not properly engineered. To this respect, authentication of the OSI layer 2 devices and its relation to the Authentication, Authorization, and Accounting (AAA) server is very important.

Similarly to data storage, the strongest requirement that the Smart Grid security subsystem must fulfill is the need to continue operating even upon temporary communication disconnections leading to network partitions. This suggests the usage of dis-

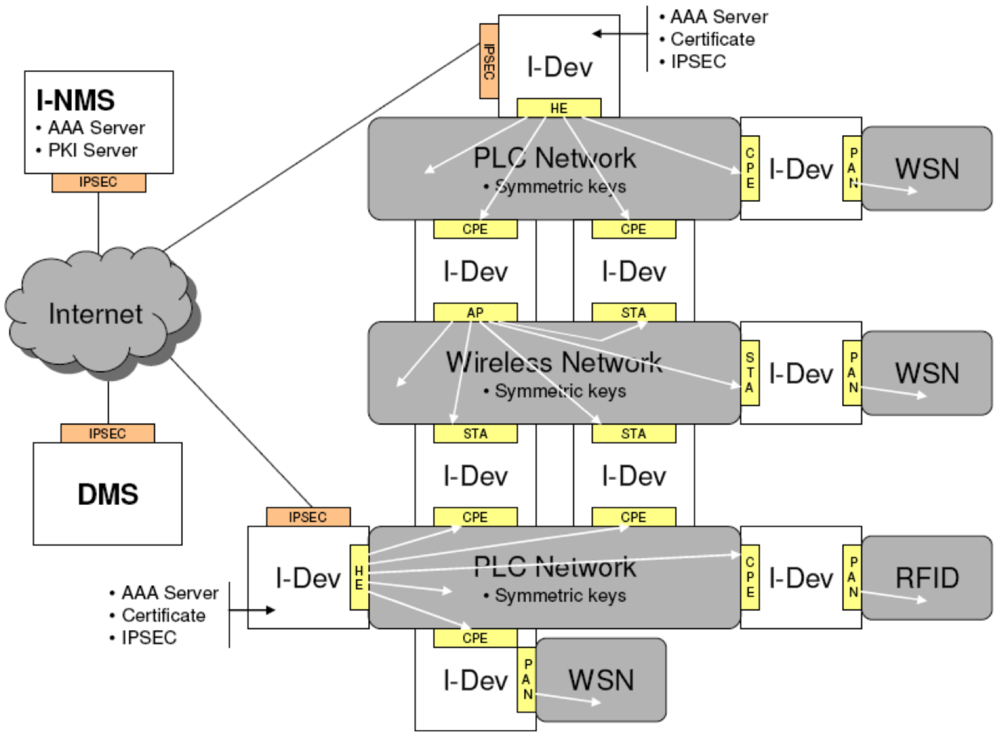
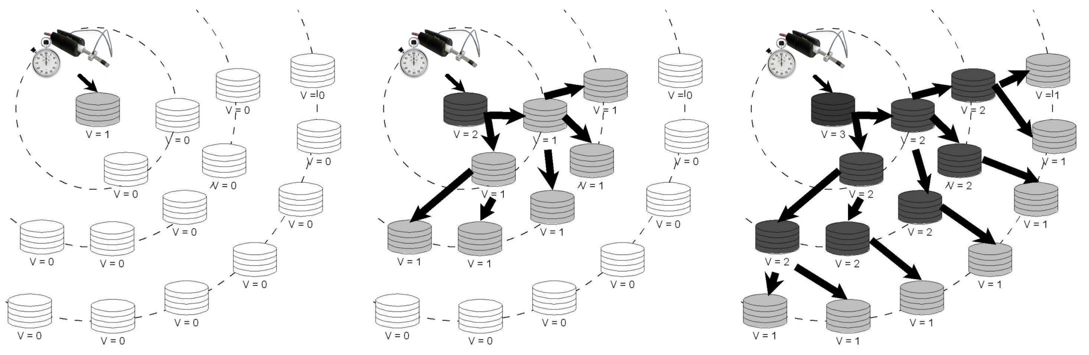


Figure 6.2: Security diagram for the INTEGRIS platform and location of the security servers (I-NMS: INTEGRIS Network Managemet System. PKI: Public Key Infraestructure Server. AAA Server).

tributed security techniques—rarely found in current security infrastructures [Carcano et al., 2011]—to avoid the typical issue of the single point of failure. A straight-forward example of security distribution can be found in the AAA server, which needs to be deployed adequately placed at different locations along the Smart Grid.

In this regard, our proposed distributed security module for the Smart Grid (see Figure 6.2) outreaches the specifications claimed by IEC62351-6 [IEC, 2007a] as it covers (1) certificates and symmetric keys, (2) AAA system (i.e., RADIUS and Diameter protocols), (3) cryptographic protocols (i.e., TLS and IPsec [Kent and Seo, 2005]), (4) encryption mechanisms, and (5) authentication mechanisms.

Overall, we have proposed a transparent multi-level hybrid security system spanning the whole Smart Grid both vertically—at different protocol layers—and horizontally—at different I-Devs. It is worth mentioning that during the development of this security system, we have observed that the security layer may limit the scalability and performance of the distributed system, due to the application of too restrictive security policies. Besides, the security system interacts with the cognitive system and contributes to the global optimization of the INTEGRIS system, by providing it with relevant security metrics, as discussed later in this chapter. As a result, other subsystems of the Smart Grid's ICT layer (i.e., distributed storage and cognitive systems) can use this secured communications architecture and inherit all its benefits.



(a) Initial state. I-Dev (gray) gathers data from IEDs.

(b) First layer update of the replication process.

(c) Second layer update of the replication process.

Figure 6.3: A given I-Dev replicates data from its attached IEDs with a replication depth of 3. The process of propagating updates in the system. As the time goes on new versions of data (V) are propagated through neighboring I-Devs.

6.3 INTEGRIS DISTRIBUTED DATA STORAGE SYSTEM

Due to the importance of the service they are delivering, Smart Grids demand a highly reliable metering and monitoring storage infrastructure. In fact, Smart Grids do not fit any more in the centralized landscape that traditional power networks suggest; there is not a single point of monitoring (e.g., power consumes can be real-time-audited at home), there is not a single point of power generation (e.g., solar panels or wind turbines deliver some power to the electric network), and obviously there is the need to improve the features of classic electric networks from a business model point of view (see Chapter 5). Therefore, Smart Grids, as an evolution of traditional power electric networks, propose a new challenging and appealing scenario to exploit the benefits of distributed systems.

As discussed in Chapter 5, the large number of IEDs (and I-Devs) spread all over the Smart Grid has rocketed the amount of data generated by electric networks, which prevents practitioners from using poorly scalable and non-elastic [Serrano et al., 2007; Curino et al., 2011b] classic relational databases (see Chapter 3). This leads us to think about NoSQL [DeCandia et al., 2007; Chang et al., 2008; Cooper et al., 2008] alternatives that typically resign to the semantic richness of the relational algebra in favor of scalability and elasticity by (1) decoupling data schemas (i.e., converting them to key-value stores), (2) relaxing data consistency and, thus, their ACID properties, and (3) providing high availability with the goal to scale up to (ideally) infinite (see Chapter 4). Although Smart Grids may fit pretty well in this paradigm—in terms of massive data storage requirements—the cost of deploying them over a Smart Grid is too expensive given the reduced computing facilities of I-Devs [INTEGRIS, 2011]. In addition, these general-purpose designed software packages prevent us from having a full control of the data storage bottom layer (i.e., replication process, concurrency management, data placement).

Therefore, we propose to deploy the custom key-value distributed storage system—and its associated data epidemic replication protocol whose behavior is summarized in Figure 6.3—detailed in Chapter 5 to address the storage requirements posed by the Smart Grid.

Despite the already exhibited advantages of this approach, it is not trivial to find out the optimal value of replication depth nor freshness level for each data item within the real-world Smart Grid. Actually, the (1) vast amount of I-Devs and IEDs, (2) large variety of information sources, and (3) different data access patterns generated by smart functions, make the task of configuring this distributed data storage repository overwhelming [Navarro et al., 2013]. Thus, we rely on a superior—but still at the Smart Grid's ICT layer—entity with an updated knowledge of the whole Smart Grid (i.e., network and routing status, security policies, data storage system behavior) to dynamically configure our system: the INTEGRIS cognitive system.

6.4 THE COGNITIVE SYSTEM OF INTEGRIS

Efficiently monitoring and managing the variety of technologies associated to the subsystems residing at Smart Grid's ICT layer (i.e., communications network, security, and data storage) is a major challenge [Monti and Ponci, 2010] due to (1) the resource limitations (e.g., time, computing facilities, network bandwidth) and (2) the necessity of dealing with such an heterogeneous and ever-changing environment.

In this regard, we have introduced the idea of using an intelligent cognitive-based system with a certain degree of awareness about its surroundings. This kind of systems automatically learn from the experience and infer patterns, behaviors, or conclusions from data by modeling the unknown structure of the problems they face [Holland, 1992]. The main difference with respect to individually monitoring and managing each subsystem is that this cognitive approach integrates diverse information from the whole ICT layer of the Smart Grid. Upon this global information, it can decide to modify the local behavior of a given subsystem to improve the overall ICT performance.

6.4.1 *Machine learning and Smart Grids*

So far, machine learning techniques [Mak, 2010; Hooper, 2010; Liu, 2010] have shown to be successful on addressing the concrete challenges posed by Smart Grids (e.g., Timed Automata based Fuzzy Controllers for autonomous voltage regulation [Acampora et al., 2011], intelligent distributed intrusion detection system [Yichi Zhang et al., 2011]). Also, these techniques—specifically Learning Classifier Systems (LCSs) [Holland, 1992]—have been used to build cognitive systems. For instance, Carse et al. [1996] applied a fuzzy Pittsburgh LCS to distributed adaptive routing control, which improved the results of a simulated telecommunication network compared to the traditional routing methods such as Dijkstra and Bellman-Ford algorithms. Also, Preen [2009] enhanced the XCS [Wilson, 1995] to forecast financial time series and obtaining interesting results. Later, Wong and Schulenburg [2007] adapted the XCS to build a trading system in a multi-agent fashion, which outperformed typical benchmark agents. Finally, Bull et al. [2004] used the XCS to build an efficient distributed adaptive control for a road traffic junction signaling under different scenarios and using a variety of parameters.

Despite the benefits on using intelligent systems for controlling specific subsystems of the Smart Grid [Mak, 2010; Hooper, 2010; Liu, 2010], there are no practical contributions on building a cognitive system to address it as a whole [Venayagamoorthy, 2011]. Therefore, we propose to build a cognitive system based on XCS to automatically control and configure the Smart Grid. We have selected XCS due to (1) its incremental

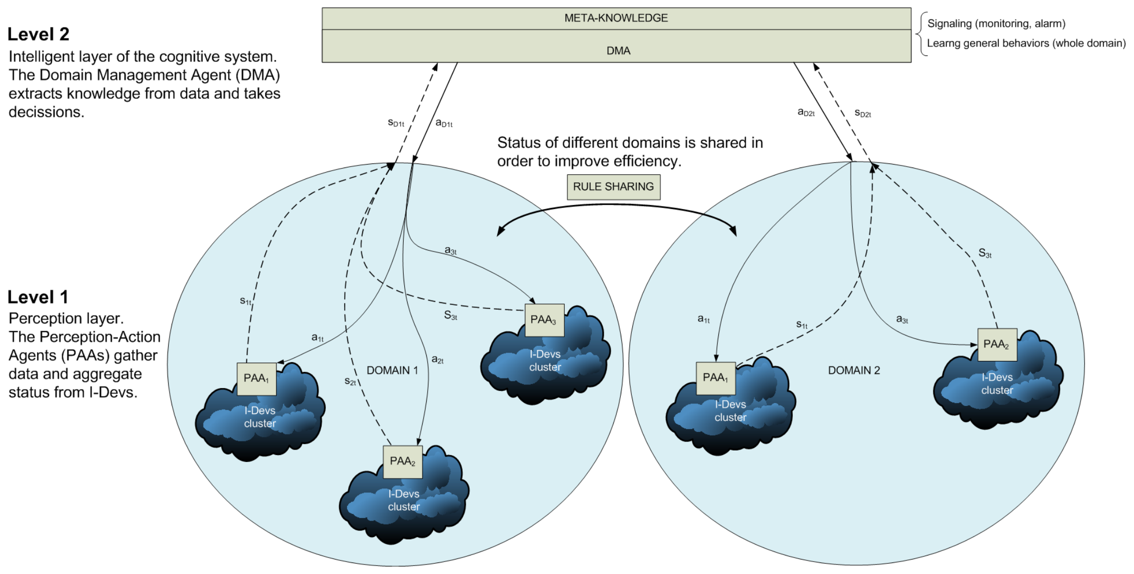

Level 2
Intelligent layer of the cognitive system. The Domain Management Agent (DMA) extracts knowledge from data and takes decissions.
META-KNOWLEDGE
DMA
Signaling (monitoring, alarm)
Learng general behaviors (whole domain)
s_{D1t}
a_{D1t}
a_{D2t}
s_{D2t}
Status of different domains is shared in order to improve efficiency.
RULE SHARING
Level 1
Perception layer. The Perception-Action Agents (PAAs) gather data and aggregate status from I-Devs.
a_{3t}
s_{1t}
S_{3t}
a_{1t}
PAA$_3$
I-Devs cluster
DOMAIN 1
a_{2t}
s_{2t}
PAA$_1$
I-Devs cluster
PAA$_2$
I-Devs cluster
S_{3t}
a_{1t}
s_{1t}
a_{3t}
PAA$_2$
I-Devs cluster
PAA$_1$
I-Devs cluster
DOMAIN 2

Figure 6.4: Schema of the cognitive system depicting two INTEGRIS I-Domains. PAA compose the perception layer, gathering information around the I-Domain. The DMA gives the intelligence of the whole scenario by taking decisions.

learning nature, which allows the system to directly learn from data streams, (2) the robustness of XCS to noisy data [Butz, 2006], (3) the transparency and generalization of the model produced by XCS in form of human-readable production rules [Butz, 2006], and (4) the fact that it has been tested in similar environments proving that it can perform properly in dynamic situations. The architecture of the developed proposal is reviewed in what follows.

6.4.2 *Cognitive system architecture*

The INTEGRIS cognitive system is devised as a global system that perceives the general state of the Smart Grid's ICT layer and decides how the different subsystems must be configured in order to maximize the overall performance.

In an ultimate attempt to improve the scalability of the cognitive system and reduce the number of attributes to speed up the learning process, we have split the XCS schema into two layers; (1) the Perception-Action Agents (PAAs), and (2) a Domain Management Agent (DMA). Additionally, we have also taken advantage of the logical layout proposed in Chapter 5 concerning I-Domains—blue circles in Figure 6.4—, which allows the system to perform parallel learning. That is, the system regularly shares rules between I-Domains to find the best possible control model, as suggested in [Bull et al., 2007]. Inside each I-Domain, there is a PAA for each cluster of I-Devs—dark blue clouds in Figure 6.4—, which allows to gather a fine knowledge from each Smart Grid segment.

PAAs are those in the bottom layer of the architecture depicted in Figure 6.4 and have a limited perception of their associated I-Domain. Specifically, these agents have the following characteristics:

- **Placement.** They are physically placed at I-Devs.
- **Perception.** They perceive the state of their PAA neighbors through the DMA.

Metric	Subsystem	Description
Bandwidth inversion	QoS - Network	Communication links cost.
Degree of connectivity	QoS - Network	Number of links towards the control center (reliability).
Degree of congestion	QoS - Network	Lengths of queues and discarding rates.
Security score	Security	Sum of several security indicators related with Integrity, Confidentiality, Authentication and DoS attacks.
Access delay	Distributed storage	Response time of the queries to the replicated data.
Global ICT layer performance	Cognitive	System performance prediction made by the cognitive agents.

Table 6.1: Selected metrics to infer the Smart Grid's ICT layer performance.

- **Tasks.** Each PAA reports to the corresponding DMA. Then, this DMA can force a given PAA from its I-Domain to apply any action (also referred to as I-Domain policy) regarding the security, communications, or storage subsystems.

  Specifically, PAAs collect meaningful information from the security, network communications, and distributed subsystems (see Table 6.1) in order to come up with a suitable policy for the whole I-Domain. Recall that this information is already known by I-Dev and, thus, its recollection does not produce an extra overhead.

Overall, PAAs are the simplest agents of the architecture, with a limited visibility of the network that contribute in the routing and quality of service monitoring and management.

The DMA is responsible for carrying out all the machine learning operations within the I-Domains. Therefore, the DMA perceives the state of all domain nodes and, if necessary, applies actions that individually or generally affect the PAAs. Specifically, the DMA has the following characteristics:

- **Placement.** It is placed external to the I-Domains: close to the Smart Grid's application layer.
- **Perception.** All the I-Domains. The state of the system will be supplied by each individual PAA.
- **Tasks.** It contains the proposed XCS algorithm. It is responsible for exchanging information status about PAAs and balancing the tradeoff between the level of security, the communications network performance, and the distributed storage actions.

Specifically, upon all the information delivered by PAAs it computes the Global ICT layer performance metric (see Table 6.1), that estimates the Smart Grid's near future whole status. It is worth mentioning that this integration model has never been applied before to SmartGrid networks.

6.4.3 *XCS overview*

XCS, the most studied LCS, evolves a population [P] of classifiers. Each classifier contains a production rule which takes the form [Wilson, 1999]:

$$\textbf{if } x_1 \in [\ell_1, u_1] \wedge x_2 \in [\ell_2, u_2] \wedge \cdots \wedge x_k \in [\ell_k, u_k] \textbf{ then } a_j,$$

where the leftmost part contains k input variables that take values of the interval $[\ell_i, u_i]^k$, and the rightmost part denotes the predicted action a_j. The parameters used by a classifier are the following: (1) an estimate of the reward the system will receive if the advocated action a_j of the rule is selected as output, namely p, (2) the error in the prediction ϵ, (3) the fitness F of the rule, (4) the experience exp, (5) the numerosity num or number of copies of this particular classifier in [P], (6) as, an estimate of the average size of the actions sets in which the classifier has participated, and (7) the timestamp ts of the classifier.

Each time the environment (i.e., PAAs) provides a new training example e, XCS creates a match set [M] of classifiers consisting on those whose conditions match the input example. A rule matches an input example if $\forall_i : \ell_i \leqslant e_i \leqslant u_i$. If [M] contains less than θ_{mna} classifiers (where θ_{mna} is a configuration parameter) with different actions, a covering mechanism is triggered which generates many different and matching classifiers (by means of using r_0 configuration parameter to tune the different intervals) as actions not previously present in [M]. For each of these ones, the system computes a prediction of the payoff to be expected and stores them in the prediction array PA. It is computed as the fitness-weighted average of all matching classifiers that specify action a. If the system is in training mode, XCS chooses the action randomly out of PA. All the classifiers in [M] advocating the selected action are put in the action set [A] and a is sent to the environment, which returns a reward ρ. Next, the parameters of classifiers in [A] are evaluated. This is performed as follows: the experience of each classifier cl is the first parameter to be updated. Next, the rest of parameters:

$$cl.p \leftarrow cl.p + \beta\,(\rho - cl.p)\,\frac{cl.F}{\sum_{cl_j \in [A]} cl_j.F}, \tag{6.1}$$

$$cl.\epsilon \leftarrow cl.\epsilon + \beta\,(|\rho - cl.p| - cl.\epsilon), \tag{6.2}$$

$$cl.as \leftarrow cl.as + \beta(\sum_{cl_j \in [A]} cl_j.num - cl.as), \tag{6.3}$$

where β ($\beta \in [0, 1]$) is the learning rate and $\sum_{cl_j \in [A]} cl_j.num$ is the size of the current action set [A]. Notice the use of gradient descent in Equation 6.1. This is done to improve the capability of the classifier prediction [Butz, 2006]. The update of the classifier fitness F is done in several steps. First, for each classifier, the accuracy K is computed:

$$cl.K \leftarrow \{ \begin{array}{ll} \alpha(cl.\epsilon/\epsilon_0)^{-\nu} & \text{if} cl.\epsilon \geqslant \epsilon_0; \\ 1 & \text{otherwise.} \end{array} \tag{6.4}$$

Where ϵ_0 is a configuration parameter which indicates the error threshold under which the accuracy of a classifier is set to 1 and α and ν are other related configuration parameters. After this, the relative accuracy K' is computed for each classifier:

$$cl.K' \leftarrow \frac{cl.K \cdot cl.num}{\sum_{cl_i \in [A]} cl_i.K \cdot cl_i.num}. \tag{6.5}$$

Finally $cl.K'$ is employed to update the fitness:

$$cl.F \leftarrow cl.F + \beta(cl.K' - cl.F). \tag{6.6}$$

XCS uses a steady-state GA to discover new rules. This last mechanism is triggered on [A] when the average time since its last application exceeds θ_{ga} (where θ_{ga} is a configuration parameter). If it is triggered, the GA selects two parents from the current [A] using tournament selection with size τ (a configuration parameter). The two parents are copied into offspring and then crossover and mutation operations are performed to these with probabilities P_χ and P_μ respectively. BLX-α is used as crossover operator due to its advantages with respect to classic two-point crossover mechanism [Morales-Ortigosa et al., 2009]. Next, fixed interval mutation is used, adding or subtracting a random quantity in the range $[0, m_0]$ to one or both boundaries of each variable interval. The predicted action of the offspring follows also this process of mutation, with probability P_μ. Parents stay in the population competing with their offspring. These are introduced to [P] via subsumption: if there exist an experienced classifier ($cl.exp \ \theta_{sub}$, where θ_{sub} is a configuration parameter) and accurate ($cl.\epsilon \ \epsilon_0$) in [A] whose condition is more general than the new offspring, the numerosity of this classifier is increased and the offspring is discarded. Exceeding classifiers are deleted form [P] with probability directly proportional to their action set size estimates. That is:

$$cl.p_{del} \leftarrow \frac{cl.d}{\sum_{\forall cl_i \in [P]} cl_i.d}, \tag{6.7}$$

where

$$cl.d \leftarrow \{ \begin{array}{ll} \frac{cl.num \cdot cl.as \cdot \bar{F}}{cl.F} & if cl.exp \ \theta_{del} \wedge cl.F \ \delta\bar{F}; \\ cl.as \cdot cl.num & otherwise, \end{array} \tag{6.8}$$

where $\bar{F}$ is the average fitness of [P], δ is a threshold and θ_{del} is the deletion threshold.

To further improve the performance of the system, at the end of each learning iteration and after the GA has been triggered, an online covering operator [Orriols-Puig et al., 2010] is applied, generating a new matching classifier with the correct action a_c if the prediction of this action was not present in the PA. To do so, a new user defined parameter is introduced, c_0, which determines the generality of the variable conditions of the new classifier.

When XCS is in the test phase—after training—the action inference is performed using the current knowledge acquired during the training stage. This process is performed in the following way: a new example is given to the system and all the matching classifiers in [P] vote for the action they predict. The most voted action is returned as the output (i.e., Global ICT layer performance).

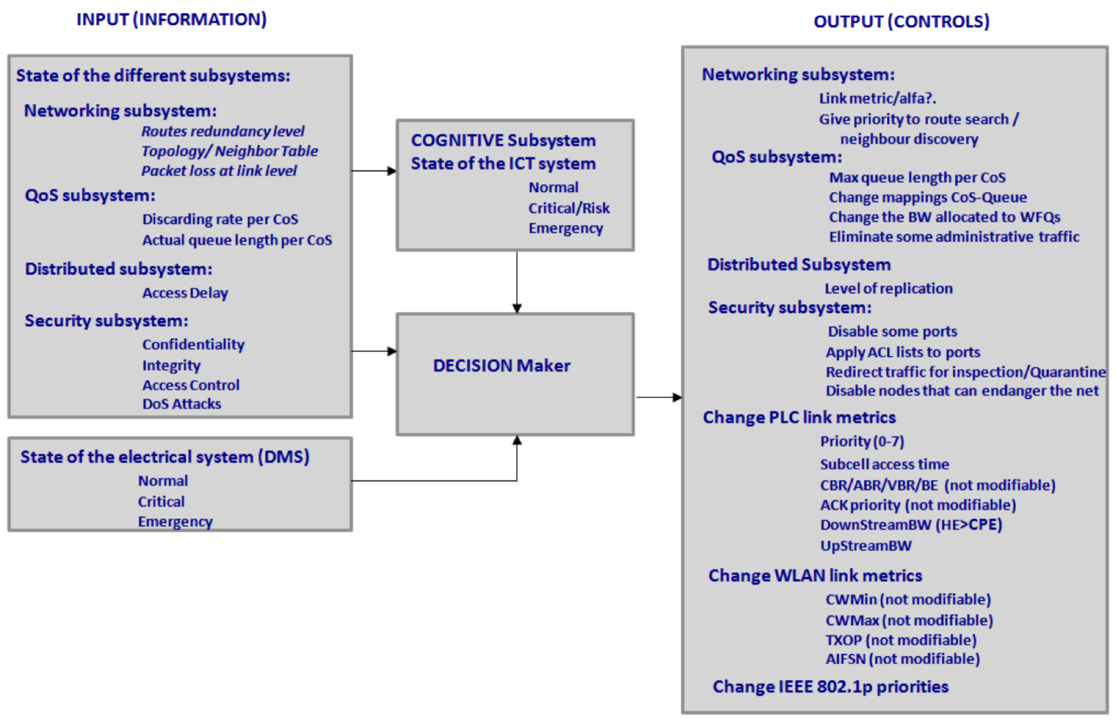


Figure 6.5: Security, network communications, distributed storage and cognitive subsystem interactions at the Smart Grid's ICT layer.

6.5 INTEGRATED PROPOSAL AND EXPERIMENTAL RESULTS

Figure 6.5 shows the interactions between all the subsystems at the ICT layer. Specifically, it links the metrics collected of each subsystem (i.e., input information) with the available actions (i.e., output controls) that allow driving the Smart Grid to a desired state. Such relation is made thanks to the aforesaid cognitive subsystem and aided by a policy-based decision system that takes into account (1) the current ICT status, (2) the predicted ICT status, and (3) the electric layer status. For instance, if the access delay is low but the queues length is high, the decision maker may decide to reduce the level of replication, that is, the replication depth (see Chapter 5).

System integration is one of the major challenges when proposing these kind of solutions. Actually, simulation tools are not suitable in this situation because they are unable to handle the variety of functions finally conducted by I-Devs:

- **Security subsystem.** Handle security related to I-Devs access control (e.g., authorized users list), network communications (e.g., asymmetric certificates), and data storage (e.g., encryption).

- **Communications subsystem.** Act as a low latency, highly reliable bridge when required by creating a meshed and QoS-aware Ethernet network among I-Devs. Also, they can connect this meshed network to a Wide Area Network (WAN) to reach the desired reliability level.

- **Distributed storage subsystem.** Behave as primaries, pseudo-primaries, or secondaries (see Chapter 5) to collect, store, and replicate to neighboring I-Devs data from their directly attached IEDs.

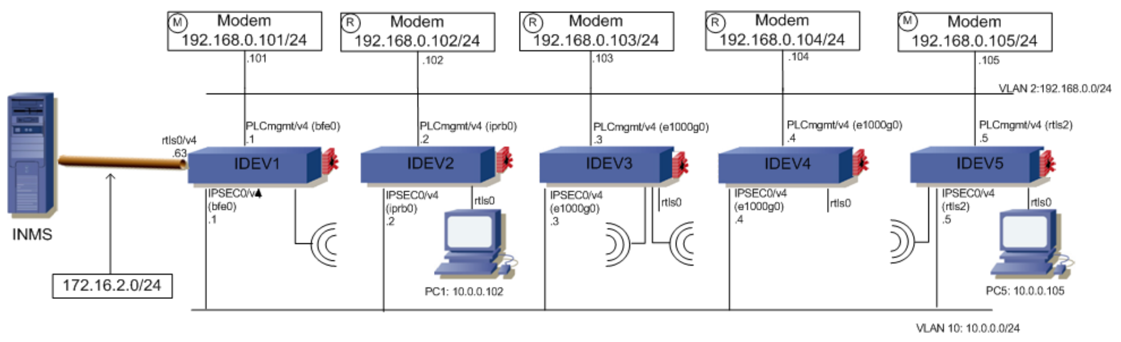


Figure 6.6: INTEGRIS testbed layout. 1 I-Domain, 5 I-Devs, and redundant communication routes through different network technologies; BB-PLC and Wi-Fi. (NMS ≡ I-NMS).

- **Electric layer.** Operate electric actuators (e.g., synchrophasors, oscilloperturbographs, etc.).

To proof the feasibility of our proposal we have conducted three experiments that are devoted to show (1) the successful integration and deployment of the proposed key-value distributed storage system at the Smart Grid's ICT layer, (2) the cognitive system ability to learn and predict the Smart Grid status, and (3) exhibit how the system reacts in advance to situations that can compromise its integrity. Specifically, Experiments I and II are implemented in a prototype scenario depicted in Figure 6.6 that models an I-Domain with 5 I-Devs. Experiment III collects the experiences when deploying an expanded version of the prototype in a real-world environment. It is worth mentioning that due to the reduced number of available I-Devs in this prototype and to speed up the development and testing process, the cognitive system has been implemented in a single machine (i.e., PAA and DMA software blocks are placed in the same I-Dev).

Status information has been gathered from the three aforementioned subsystems once a minute during 28 hours. During this period of time and using an incremental clustering technique [Xu and Ii, 2005], our experts have been able to differentiate up to 64 000 different situations or examples clustered in eight different groups that can be summarized into the three categories—normal, critical/risk, and emergency—shown in Figure 6.5. This clustering process has been done—following the work of Lu et al. [2008]—using an online K-Nearest Neighbor due to its simplicity, effectiveness, and incremental behavior, which allows a perfect integration with the cognitive system architecture. For more information about this process the reader is referred to [Lu et al., 2008]. The rationale of using an XCS remains on its ability to generalize by inferring patterns from these data, which allows taking accurate decisions by itself when previously unseen cases appear.

6.5.1 *Experiment I: Proof of concept*

In order to test the response of the cognitive system, a proof of concept experiment has been done. In this experiment the idea is to empirically characterize how the learning mechanism behaves using a classic train-test approach in which data are split into two different sets: (1) the train set which is used by our algorithm to learn and (2) the test set which is used to check the accuracy of the learner. Data consists in 11 variables describing the scenario, each collected by PAAs and delivered to DMA via network

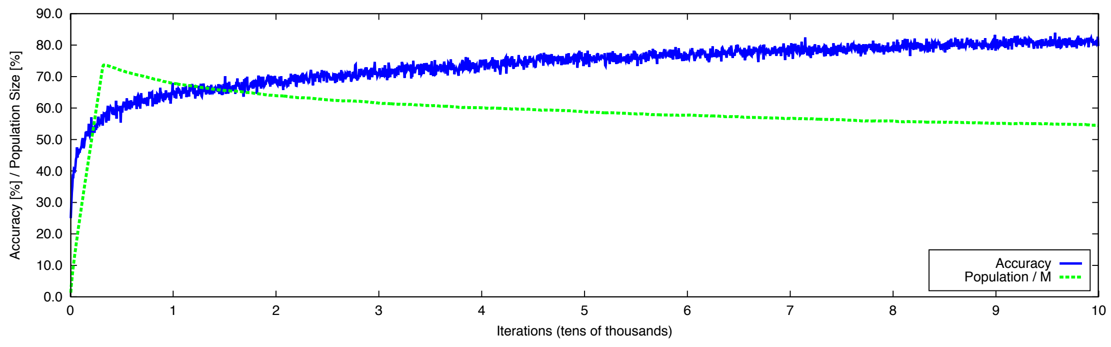


Figure 6.7: Results of the proof of concept experiment. Solid curve: performance, the fraction of last 50 exploit problems correct. Dashed curve: Population size in macro-classifiers (divided by $M = 6400$).

loopback sockets. These variables include (1) communications network metrics, (2) I-Devs level of security, (3) response time of the distributed storage system, and (4) security metrics (check Table 6.1). In addition, to further stress the system, there were eight possible actions (also referred to as labels) representing the possible status of the Smart Grid (i.e., critical, normal, emergency) obtained by the incremental clustering process.

The configuration parameters for the experiment were the following: $\alpha = \delta = 0.1$, $\beta = 0.2$, $\epsilon_0 = 1$, $\nu = 10.0$, $P_\chi = 0.8$, $P_\mu = 0.04$, $\theta_{\{ga, del, sub\}} = 15$, $\tau = 0.4$, $m_0 = c_0 = 0.2$, $r_0 = 1$, and maximum population size was set to 6400 classifiers. The system was tested using $100\,000$ iterations. The experiment was repeated with 30 different random seeds, and the results are averages of these runs.

Because we are interested in the instantaneous measurement of the accuracy of the system and this one may vary greatly under certain conditions, we used an exponential smoothing formula in order to reflect that instantaneous measurement. We used the α parameter as recommended in [Nagle, 1987] and in [Núñez et al., 2007], that is $\alpha = \frac{7}{8}$:

$$smoothedAcc(t) = \alpha \cdot smoothedAcc(t-1) + (1-\alpha) \cdot acc,$$

where $smoothedAcc(t)$ and $smoothedAcc(t-1)$ are the current and previous smoothed accuracy values and acc is the accuracy given by XCS.

Figure 6.7 shows the performance and population size averaged over 30 runs, each with a different random seed, of the experiment. This graph indicates that the problem is quite difficult to learn—as we expected—and this is visible by the large amount of iterations required for XCS to get above 80% of accuracy. The generalization capabilities of XCS are also visible in the population curve: initially the population grew very fast and, after reaching a point of equilibrium at 74%, it started to generalize by decreasing the number of classifiers while obtaining more accurate results. This experiment allows us to see that XCS is competent in this kind of environments, having a global view of the system intrinsics when integrating the different technologies.

6.5.2 *Experiment II: System dynamics*

The proof of concept experiment has shown empirically that the integration of data collected from different subsystems is possible using XCS as a learner by giving competent

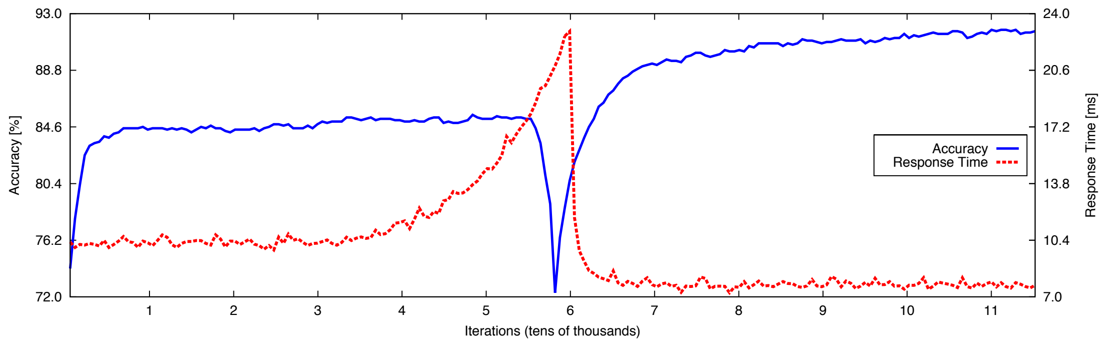


Figure 6.8: Results of the system liveness experiment. The change in concept is clearly visible at iteration 57 600. Solid curve: performance of the system. Dashed curve: response time.

results. However, this experiment has not checked the adaptability of the system under changing environments. For this purpose, a new experiment is performed using the scenario depicted in Figure 6.6 in which, at a certain point, we have intentionally introduced a considerable amount of dummy traffic to the network—considerable enough to alter system's normal operation—and, thus, force a concept change in the cognitive system.

In this experiment the stream of data lasts for 115 200 iterations and the concept is changed at iteration 57 600 by forcing a steep change in the response time of the system. To evaluate how the cognitive system is adapted to the concept change, two test sets containing each 6400 previously unseen examples are used: T_1 and T_2. The first set contains the test data of the first target concept. The second one is used to test the adaptability of the system to the second concept and contains test examples from the new target concept. T_1 (and after the forced concept change T_2) are used periodically during training to evaluate the performance of the system.

As in the previous experiment, data consists of 11 variables describing the scenario, each gathered by PAAs and delivered to the DMA via network loopback sockets, but this time the values are changed dynamically to force the concept change.

The configuration parameters for the experiment were the following: $\alpha = \delta = 0.1$, $\beta = 0.2$, $\epsilon_0 = 1$, $\nu = 5.0$, $P_\chi = 0.8$, $P_\mu = 0.167$, $\theta_{ga} = 25$, $\theta_{del} = 20$, $\theta_{sub} = 50$, $\tau = 0.4$, $m_0 = 0.2$, $c_0 = 0.167$, $r_0 = 1$, and maximum population size was set to 6400 classifiers. The accuracy of the system was evaluated every 576 iterations with one of the test sets (T_1 during the first concept and later T_2 during the second concept). As in the previous experiment this one was repeated with 30 different random seeds, and the results are averages of these runs. We also used the exponential smoothing formula.

Figure 6.8 shows the evolution of the test performance, each with a different random seed, of the experiment. Actually, this experiment can be best seen as a validation proof of our approach since it covers all subsystems: once the cognitive system realizes that the network is being saturated—which prevents distributed subsystem to replicate data properly—through the communications network subsystem, it decides that this I-Domain should enter in a different state and, thus, broadcasts all other subsystems about this decision. According to this new state and the policy based directives, the security subsystem configures I-Dev access control lists to make them drop the best-effort network traffic. Finally, the distributed system obtains a lower response time than in the previous state due to this new I-Domain configuration.

In this particular experiment the cognitive system showed a high degree of robustness, being able to perform a quick recovery after the concept change. This phenomenon is due to (1) the online learning architecture which detects and discards old classifiers when they no longer fit and (2) the effective enhancement of the system provided by the online covering operator, which generates highly fit classifiers when required and adding them to the final population.

6.5.3 *Experiment III: System deployment and integrated monitoring*

Once the previous experiments have proofed the feasibility of our approach with the hardware standing inside our lab, we decided to move it to a real-world and larger scenario (more than just 5 I-Devs). We chase an area with a high number of photovoltaic devices installed in buildings and houses. Thus we selected a Medium Voltage feeder of the distribution grid of A2A Reti Elettriche SpA in the city of Brescia (Italy), to perform the testing of the integrated monitoring use case [INTEGRIS, 2011; Della Giustina et al., 2011]. The preliminary results of real-time low voltage network management developed have been already presented in [Repo et al., 2011; Della Giustina et al., 2011]. Moreover, three low voltage network management use cases are defined and analyzed from ICT and electrical engineering viewpoints [Repo et al., 2011; Della Giustina et al., 2011].

We have seen that the defined communication system makes possible layer 2 communication over electrical distribution areas allowing the use of IEC 61850 protocols [IEC, 2003]. Moreover, the information security of the proposed system is guaranteed by the application off-the-self security protocols in the context of the Smart Grid. Also, we have observed that our custom key-value distributed storage protocol is able to successfully handle the Smart Grid's data requirements in a real-world scenario. With this experiment we conclude that our approach still works in a hostile real environment and further endorses the feasibility of our proposal.

Overall, experiments show that the integration of the different technologies is not only feasible but valuable due to the benefits it gives. It is worth mentioning that the cognitive system has shown an eagerly predictive behavior, reacting in advance to delicate situations in the Smart Grid and, thus, optimizing the system resources. However, there is still a long way to do in terms of performance assessment, since neither standard benchmarks nor reference feature values of the Smart Grid's ICT layer have been proposed so far.

As far as the distributed storage system is concerned, we have successfully reached the requirements announced in Chapter 5. However, as the Smart Grids concept evolves, smart functions complexity will probably rise [Gungor et al., 2013]. This means that a custom key-value store might not be enough to cover all the necessities posed by the applications running on the Smart Grid. Therefore, the following chapter proposes to extend this custom key-value storage architecture—inspired by cloud storage and classic databases replication techniques—and enable it to provide transactional support.

Contribution.

1. Deployment of a complete ICT architecture especially designed to meet the Smart Grid requirements.

2. Revision of a low layer communication network, which provides both security and QoS-aware routing facilities adaptable to the elastic nature of the Smart Grid.

3. Integration of the proposed custom key-value distributed data storage architecture to face a real-world problem.

4. Introduction of a cognitive system able to have a global perception of the Smart Grid, build behavior rules from the knowledge acquired, and anticipate to the Smart Grid situations.

7 PROVIDING TRANSACTIONAL SUPPORT ON THE CLOUD

Summary. Classic replication protocols running on traditional cluster-based databases are currently unable to meet the ever-growing scalability demands of many modern software applications. Typically, such limitations have been addressed by cloud storage repositories that favor availability and scalability over data consistency and transactional support. This chapter extends the key-value distributed storage architecture described in previous chapters and presents Epidemia, a hybrid approach that combines classic database replication techniques with a cloud-inspired infrastructure to provide transactional support and high availability. To prove the feasibility of the system, an analytical model for computing its scale out and a formal correctness verification are developed. Additionally, the effects of the selected data partitioning scheme and replication protocol are empirically analyzed in a prototype implementation.[6].

"The first step towards getting somewhere is to decide that you are not going to stay where you are"
— John Pierpont Morgan, 1955.

7.1 INTRODUCTION

The ambitious requirements regarding availability and fault tolerance demanded by many modern software applications entail the need for replicating and persistently storing vast amounts of data. So far, cluster-based databases have traditionally been considered as the proper choice, running either primary-backup [Daudjee and Salem, 2006] or update-everywhere [Wiesmann and Schiper, 2005] replication protocols (Chapter 2), despite their scalability limitations [Gray et al., 1996].

As shown in Chapter 3, replication is the main cause of the stringent scalability capabilities of cluster-based databases, in the sense that the greater the number of replicas that have to perform an update is, the less efficient the system becomes due to the overhead derived from propagating changes to all replicas [Gray et al., 1996]. Consequently, database clusters can not scale as long as strong consistency—that leads to additional network and database stalls [Armendáriz-Iñigo et al., 2007; Serrano et al., 2007; Pedone and Oliveira, 2009; Preguiça et al., 2010]—is maintained among replicas.

Cloud storage repositories and NoSQL approaches (see Chapter 4) have emerged as an alternative to obtain high scalability and availability at the cost of relaxing the traditional ACID properties and, thus relying on weak consistency models such as

6 An earlier and abridged version of the work reported in this chapter were published as the paper entitled "Providing transactional support on the cloud: Hybrid approaches" in the proceedings of the XX Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2012).

eventual consistency [Vogels, 2009]. Specifically, the data semantics considered in this kind of systems are commonly referred to as BASE [Fox et al., 1997; Brewer, 2012]—in opposition to the ACID properties—and prominently favor availability over consistency. Such evolution has been motivated by (1) the trade-off between data consistency, system availability, and tolerance to network partitions stated in Brewer's CAP theorem [Brewer, 2000, 2012], and (2) the fact that network partitions are the norm, rather than the exception, in systems spanning large and geographically separated infrastructures [Corbett et al., 2012].

Although the storage functionalities provided by these cloud repositories suffice for typical target applications (e.g., web indexing, multimedia storage or content delivery networks), there are still many use cases which cannot take advantage of the cloud paradigm because they are unable to resign from their transactional nature (e.g., some recently defined smart functions from Smart Grids [Gungor et al., 2013], travel-related booking services like Amadeus that have to deal with in course and already committed reservations).

Latest trends derived from NoSQL systems attempt to overcome this drawback by providing transactional support to a certain extent while meeting the principles of the cloud philosophy [Das et al., 2010b; Vo et al., 2010; Baker et al., 2011; Curino et al., 2011b; Levandoski et al., 2011; Sivasubramanian, 2012]. Typically, these solutions are targeted to specific purposes and restrict the scope of transactional support in different ways to provide highly available and elastic services. Therefore, the purpose of this chapter is to (1) review current approaches and strategies for providing transactional support in cloud systems, (2) present Epidemia: a distributed cloud storage repository—extended from the key-value architecture described in previous chapters—that provides transactional support, (3) propose an analytical model, (4) verify its formal correctness, and (5) analyze its performance in a laboratory-based scenario.

7.2 RELATED WORK AND DESIGN CHALLENGES

Leaving application developers in charge of ensuring transactional consistency [Brantner et al., 2008] is both costly and inefficient (thus only making sense for applications that rarely require to provide transactions with strong consistency guarantees [Kraska et al., 2009a]).

Therefore, the vast majority of cloud-based solutions that offer transactional support manage transactions on the server side. For instance, ecStore [Vo et al., 2010], which provides transactional semantics across multiple rows; ElasTraS [Das et al., 2010b], which is an elastic database system that supports a simplified type of transactions (the mini transactions originally defined in Sinfonia [Aguilera et al., 2009]) that are executed within one single data partition (see Chapter 4); Microsoft SQL Azure [Campbell et al., 2010] and Google Megastore [Baker et al., 2011], which support transactions over multiple records although they require these records to be co-located in a certain way; Relational Cloud [Curino et al., 2011b], which is a multi-tenant system that hosts a single database server in each physical machine; or Deuteronomy [Levandoski et al., 2011], which decomposes functions of the database storage engine kernel into a transactional component that manages transactions and a data component that caches data, knows about the physical organization, and supports a record-oriented interface with atomic operations: Amazon DynamoDB [Sivasubramanian, 2012], which supports implicit item-level transactions; Omid [Gómez Ferro et al., 2014], which implements a lock-free commit algorithm for transactions.

Overall, these strategies are aimed to address one ore more challenges that transactional support entails: workload management (i.e., monitoring resource utilization and dealing with data allocation issues), transaction management (i.e., ensuring correct execution of distributed transactions), and replication management (i.e., handling data replication across nodes to ensure data availability). Each challenge is articulated in the following.

7.2.1 *Workload management*

Cloud-based architectures feature an elastic scale-out to handle varying workloads, (i.e., scale up when the workload increases and scale down to save resources [Elmore et al., 2011; Curino et al., 2010, 2011a]), which enables an attractive pay-as-you-go business model. Workload management includes all hardware resources and algorithms devoted to determining the most adequate system configuration to maximize performance while minimizing resources usage (i.e., operational expenditures). The key concepts concerning different strategies to provide scalability in highly elastic and dynamic environments are in what follows:

Data Partitioning. An efficient way to support transactions on the cloud is to reduce the interaction among replicas to the minimum. A common solution is to partition data in such a way that as many transactions as possible can be entirely executed within one single partition, thus requiring no coordination with the rest of data partitions [Curino et al., 2010; Cheung et al., 2012; Pavlo et al., 2012]. Depending on the workload characteristics, it may also be convenient to dynamically re-split, merge or replicate certain partitions.

For instance, Relational Cloud [Curino et al., 2011b] recurs to partitioning upon detecting that a single host is unable to handle an entire database. To partition a database, the system analyzes queries to minimize the number of transactions that need to access several partitions. This is done by a module called Schism [Curino et al., 2010], which uses an offline graph-based partitioning algorithm to achieve efficient database partitions, where partitioning is done even at the table level. On the other hand, Kairos [Curino et al., 2011a] is the component responsible for monitoring and consolidating databases in Relational Cloud. This component estimates hardware resources requirements according to the submitted workload and produces an assignment of databases to physical machines. In Kairos, the consolidation is stated as a non-linear optimization program, aiming to minimize the number of servers and balance load while achieving near-zero performance degradation.

Another interesting approach for tackling partitioning issues is presented in [Pandis et al., 2011b]. This work introduces the concept of PhysioLogical Partitioning (PLP), a transaction processing technique that logically partitions the physical data accesses. To alleviate the difficulties imposed by page latching and repartitioning, PLP uses a new physical access method inspired by a multi-rooted B+Tree coined as MRBTree. The idea of PLP is further extended in [Tözün et al., 2012] to provide dynamic load balancing in OLTP systems.

Storing partitions in main memory. In-memory approaches can be used in cloud-based systems for enhancing the performance and scalability features of both SQL databases [Stonebraker et al., 2007] and key-value stores [Lakshman and Malik, 2010; Vo et al., 2010]. In this kind of solutions, each replica maintains all its assigned partitions in main memory, therefore avoiding intensive writing to disk [Stonebraker et al., 2007] plus saving disk stalls and the cost of a distributed storage [Das et al., 2010b]. In this case, the replication degree needs to be high enough to ensure durability. Other systems such as Lakshman and Malik [2010]; Vo et al. [2010] rely on in-memory data structures to reduce response time, but periodically dump this information to disk to ensure durability.

On the other hand, the thread-to-data policy has been shown to be effective through exploiting the regular pattern of data accesses [Jones et al., 2010; Pandis et al., 2011a]: there exists one thread per partition that executes transactions one after the other, as there is no gain from multi-threading transactions (i.e., there is no need for concurrency). Together with this, the use of stored procedures avoids any interaction with the user during the execution of a transaction [Cheung et al., 2012], hence removing client stalls. Thus, single-partition transactions can be trivially serialized [Aguilera et al., 2009; Curino et al., 2010; Das et al., 2010b; Jones et al., 2010; Vo et al., 2010].

Live migration. Live migration is motivated by the inherent elasticity of cloud environments: it consists in performing a data migration process (which might be motivated by cost-saving or performance considerations) from one or more servers to another, while interfering with other operational processes as little as possible [Das et al., 2011; Elmore et al., 2011]. Some representative approaches that address this issue are Zephyr [Elmore et al., 2011], dedicated to the live migration of shared-nothing transactional databases; and Albatross [Das et al., 2011], which delves into the case where data is stored in a network attached storage.

7.2.2 *Transaction management*

Keeping several geographically distant data partitions is very expensive in terms of performance degradation [Gray et al., 1996]. Therefore, maximizing the proportion of single-partition transactions is essential to reduce the overhead associated to transaction management. Indeed, it is well known that standard TPC benchmarks, such as TPC-C [Jones et al., 2010; Das et al., 2010b; Pandis et al., 2011a] or TPC-E [Curino et al., 2010], can be split so that every transaction accesses only one partition as already done by Curino et al. [2012].

However, there may be cases, especially in dynamic environments such as cloud applications, in which some transactions (named multi-partition transactions) need to access several partitions. The issue here is twofold: on the one hand, network stalls appear since transactions are fragmented [Jones et al., 2010] and different fragments are executed at different partitions; and, on the other hand, some coordination has to be provided to commit a transaction and, thus, maintain global consistency across partitions.

With regard to the first issue, some solutions choose to speculatively execute transactions (without notifying the clients to avoid cascading aborts [Bernstein et al., 1987]) that are ordered after a fragment that is waiting for its commit, as done by Jones et al. [2010], or to use transaction flow graphs to determine rendezvous points along the fragments of a multi-partition transaction [Curino et al., 2010; Pandis et al., 2011a]. With regard to the second issue, several techniques have been proposed so far: single coordinator [Jones et al., 2010], multiple coordinators [Maia et al., 2010], the Paxos algorithm and its variants [Lamport, 1998, 2006; Marandi et al., 2010; Corbett et al., 2012], the two-phase commit rule [Bernstein et al., 1987] and its variants [Aguilera et al., 2009; Curino et al., 2010; Vo et al., 2010] or rendezvous protocols [Pandis et al., 2011a].

7.2.3 *Replication management*

In general, cloud databases have put aside classic techniques concerning replication to achieve high scalability levels (see Chapter 2). Therefore, data is partitioned and not replicated in all replicas. Instead, data is replicated up to a given K level [Curino et al., 2010; Das et al., 2010b; Jones et al., 2010; Maia et al., 2010; Vo et al., 2010]; i.e., there exist up to K physical copies of a certain partition.

The most common replication technique used is the primary-backup mechanism using either an optimistic approach [Aguilera et al., 2009; Vo et al., 2010] (sending the reply back as soon as it is executed in the primary, and updating backups in the background) or a pessimistic one [Jones et al., 2010]. Replication can also be done by means of state machine replication [Aguilera et al., 2009; Maia et al., 2010] or Paxos [Lamport, 2006; Aguilera et al., 2009; Marandi et al., 2010]. Another approach consists in relying on a fault-tolerant distributed storage system to ensure data availability, thus delegating replication to the lower level of its architecture as done by Das et al. [2010b].

On the subject of which partitions should be replicated and to what level, there are different alternatives. In the case of read-only transactions, a possible solution is to replicate their associated partitions in all replicas to exploit the benefits of access locality [Vo et al., 2010]. As for transactions that update data, the replica placement policy follows different approaches that can be defined by means of graph analysis [Curino et al., 2010], histogram analysis of accesses to a given range, or autonomously through monitoring the workload until the K level is met [Vo et al., 2010].

In what follows, the design rationale to build Epidemia and face these challenges is elaborated.

7.3 DESIGN RATIONALE

As shown in Chapter 3, database replication protocols perform differently depending on the workload characteristics. For instance, a read intensive application will probably obtain a higher throughput with a primary-backup—that will limit the throughput of update operations [Gray et al., 1996]— scheme than with update-everywhere. In contrast, a database whose items are frequently updated might benefit from an update-everywhere replication strategy based on total order broadcast [Wiesmann and Schiper, 2005]—although it will only scale up to a certain limit, as the cost of propagating updates in a correct way in order to maintain data consistency increases with the number of involved replicas (both in terms of the number of messages to be exchanged and of concurrency control overhead).

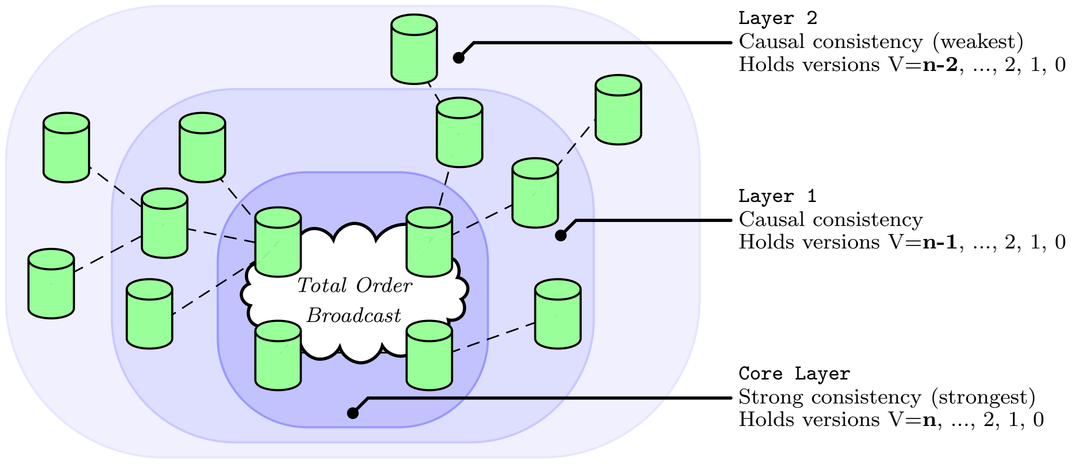

Layer 2
Causal consistency (weakest)
Holds versions V=**n-2**, ..., 2, 1, 0
Layer 1
Causal consistency
Holds versions V=**n-1**, ..., 2, 1, 0
Core Layer
Strong consistency (strongest)
Holds versions V=**n**, ..., 2, 1, 0
Total Order Broadcast

Figure 7.1: Epidemic propagation of updates.

As suggested in the design of the custom key-value distributed storage (see Chapter 5), this scalability problem could be alleviated if some (let us say M) of the replicas involved in the update-everywhere replication protocol acted as primaries for other backup replicas ($M - K$), which would asynchronously receive updates from their respective primaries. At the same time, backup replicas could act as primaries for other replicas, thus creating a hierarchy where updates would be propagated in an epidemic way. In other words, only M out of K replicas storing a partition (probably with $M \ll K$) would participate in the update-everywhere replication protocol, whereas the rest would form a hierarchy of pseudo primary-backups. Also, if the replication degree of a given partition is augmented, it becomes possible to forward transactions to different replicas, and thus, transactions will be more likely to obtain stale, though consistent, snapshots.

Therefore, each partition can be seen as a multilayer set of different consistent versions, which is very similar to the design proposed in Chapter 5. Hence, replicas closer to the core have the most recent values for data items, whereas lower levels have older versions of the data items as shown in Figure 7.1, which depicts an example of a update-everywhere partition (i.e., $M = 4$) with $K = 14$ replicas. Initially, a client executes a transaction that updates the core partition that will be propagated to other members of the partition with version $V = 0$. As the time goes by, this version will be causally propagated—using either active or passive replication—to subsequent partitions. Meanwhile, the client can execute another update transaction and its associated version increases.

Depending on the consistency level demanded for a transaction, its operations can be forwarded to replicas containing fresh or stale versions. The main contribution of this novel architecture resides in the way partitions are built. Since the replicas of each partition are organized as a hierarchy where updates are asynchronously propagated from one level to the next, clients can access different versions of the same partition. Actually, versions can be associated with timestamps, so that transactions can execute queries stating the level of freshness of the returned data [Lomet et al., 2012], using techniques to find the appropriate consistency version according to the requested timestamp [Ports et al., 2010], either by quorums [Vo et al., 2010] or by a direct request to the core layer that can be temporarily cached [Sciascia et al., 2010].

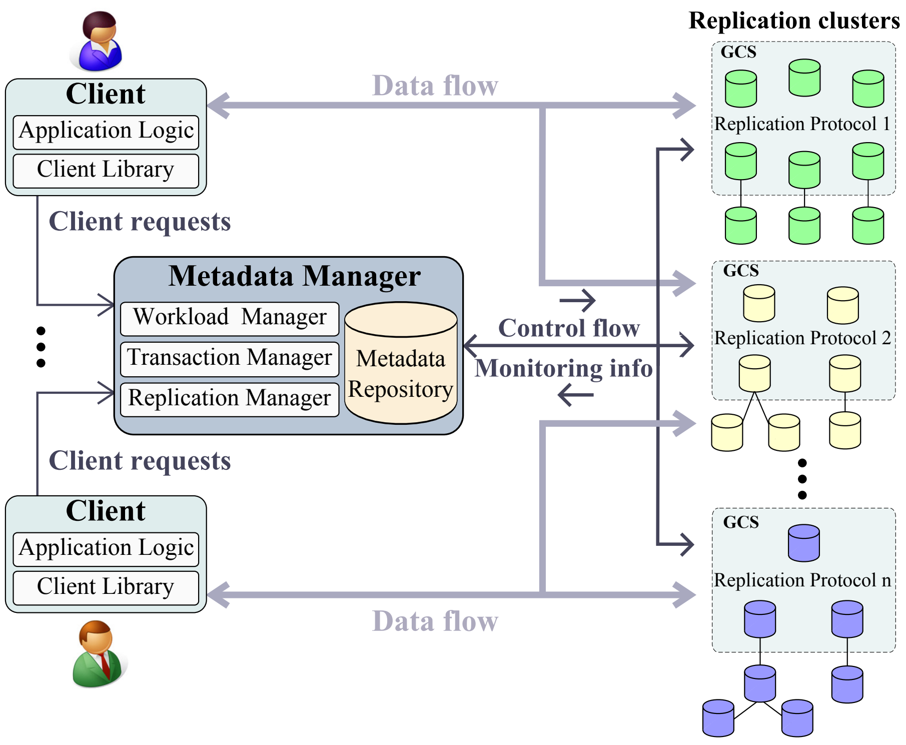


Figure 7.2: Epidemia's system model.

Another advantage of this architecture is that system replicas can be upgraded or downgraded in the hierarchy according to current system requirements. This configuration also reduces the impact of the addition of new replicas on the system's overall performance, since a new replica can start in the lowest level of the hierarchy of a partition (after performing an initial live migration [Elmore et al., 2011]) and be progressively upgraded depending on the needs of the system—which is slightly different than the recovery protocol proposed in Chapter 5.

Overall, Epidemia consists of a dynamic set of cluster-based databases (that might be hosted in a cloud infrastructure), each maintaining a hierarchy of versions where the topmost level of the hierarchy holds the newest version and consists of a set of replicas that are controlled by a classic replication protocol (see Chapter 2) that is determined depending on the current workload characteristics. The rest of hierarchy levels are updated in an epidemic way by asynchronously propagating updates from one level to the next (very similar to the strategy used in Chapter 5). Thus, depending on the consistency level demanded by a transaction, it can be forwarded to replicas containing newer or older versions. Therefore, Epidemia is able to offer a broad range of QoS levels by varying the tradeoff between consistency and availability guarantees.

7.4 SYSTEM ARCHITECTURE

Keeping the basics of the cloud-inspired key-value repository presented in Chapter 5 (i.e., having a tree of consistent versions for each data partition) as a starting point, we propose Epidemia, a distributed storage system that exploits data replication to provide a highly available and elastic database service with transactional support.

The communication among the different components of Epidemia is asynchronous and performed via message-exchange. Therefore, the system is partially synchronous in the sense that, although time bounds on message latency and processing speed exist, they are unknown [Dwork et al., 1988], which prevents system components from precisely determining whether another component that appears to be unresponsive has crashed or happens to be very slow. Also, to ensure that that messages sent from one process to another are neither lost nor reordered with respect to the order in which they were sent, we assume FIFO quasi-reliable point-to-point channels [Schiper, 2006].

These premises lead to the system model depicted in Figure 7.2, which is a particularization of the generic reference model architecture for cloud storage discussed in Chapter 4. Specifically, Epidemia own the following components:

System Clients. Client applications interact with the storage system by means of a custom client library that acts as a wrapper for the management of connections with both the metadata manager and the replicas that serve the transactions.

To minimize system stalls, interactive transactions are not supported (i.e., there is no interaction with the client during the execution of the transaction), in the sense that all the operations of a transaction are known upon submitting the transaction, and therefore they are sent and processed together (as it usually happens in stored procedures [Cheung et al., 2012]). Similarly to the case of the proposed key-value store, transactions in Epidemia can demand a specific freshness degree that can be established in terms of absolute values, version numbers, or timestamps.

In order to submit a transaction, the client application invokes the aforementioned library, which sends a request that includes the parameters of the transaction to one of the metadata manager nodes.

Upon receiving a client request, the metadata manager (1) determines the partition involved in the transaction, (2) selects one of the replicas of the replication cluster storing each of the partitions participating in the transaction, and (3) sends back the address of this delegate replica to the client. Note that the election of the most suitable delegate replica (or replicas) takes into account several factors, such as the transaction type (read-only vs. update), the current workload of the replicas or the freshness degree demanded by the transaction.

The client library also deals with failed requests to metadata manager nodes (by sending a request to a different metadata manager node in case a timeout expires without a request being answered), as well as with failed requests to replicas (by sending another request to the metadata manager in order to obtain the address of another replica that can execute the transaction). Obviously, the system will have to ensure that each transaction is executed at most once, even if the client retransmits it several times.

Metadata manager. The metadata manager has encompasses a workload manager, replication manager, a transaction manager, and a metadata repository. Fundamentally, It is in charge of maintaining the metadata repository, which contains the following information:

1. A mapping between each data item and the partition it is stored in, which can be established using different granularity levels depending on the partitioning

scheme, e.g., by associating each table to a data partition or by horizontally splitting subsets of items belonging to the same table into different partitions.

2. A set of available replicas. Note that replicas can be added (or removed) to Epidemia by modifying this data structure on the fly to inform the metadata manager of the addresses of the available resources, so that it can decide on how to use them.
3. A mapping between each data partition and the set of replicas that belong to the replication cluster that handles the partition. For each replica, it is also necessary to store the level in the hierarchy of versions it is located in.
4. A mapping between each partition and the replication protocol running on the core level of the corresponding hierarchy.
5. Status of each replica, including parameters such as number of transactions per second executed, average number of pending transactions, rate of read-only transactions executed or computing processing usage levels.

The information contained in the metadata repository has stringent consistency and availability requirements. Nevertheless, other requirements such as scalability or elasticity are not that essential, since the information stored in the metadata repository is relatively small and less frequently updated in comparison with the data of the applications stored in the system. For this reason, the metadata manager can be distributed among a small set of nodes and synchronized using Paxos [Lamport, 1998] to provide fault tolerance while ensuring data consistency and leveraging system scalability.

Replication clusters. A replication cluster consists of a set of replicas organized into a hierarchy of levels. The replicas of the core level make use of a replication protocol—selected by the metadata manager according to load carachteristics—to propagate updates to the rest of replicas of the same level. The replication protocol is built on top of a GCS [Chockler et al., 2001], which allows replicas of the core level of each partition to safely propagate their changes among themselves (see Chapter 2). In contrast, the replicas of the rest of the hierarchy levels act as mere backups for replicas of the higher level and communicate with their respective primaries using point-to-point channels.

With the aim of exploiting the advantages of in-memory approaches (see Section 7.2.1), such as avoiding disk stalls and using the thread-to-data policy, we assume that every replica keeps all its data in main memory (i.e., all the data items managed by each replica are stored in main memory). Consequently, replicas are stateless in the sense that, in case a replica leaves the system (either because it has crashed or due to a decision of the metadata manager) and then joins again, it must obtain all the data items of the partition it belongs to. This is done by transferring the whole state from one or more replicas to the new one. In case the requirements of a client application demanded stringent durability guarantees, this solution could be adapted to force replicas to transfer their data to a persistent storage device, either on a regular basis or upon receiving an order from the metadata manager.

The interactions between these three modules are exemplified in Figure 7.3. Specifically, the three graphics show the evolution across time of (a) number of requests per

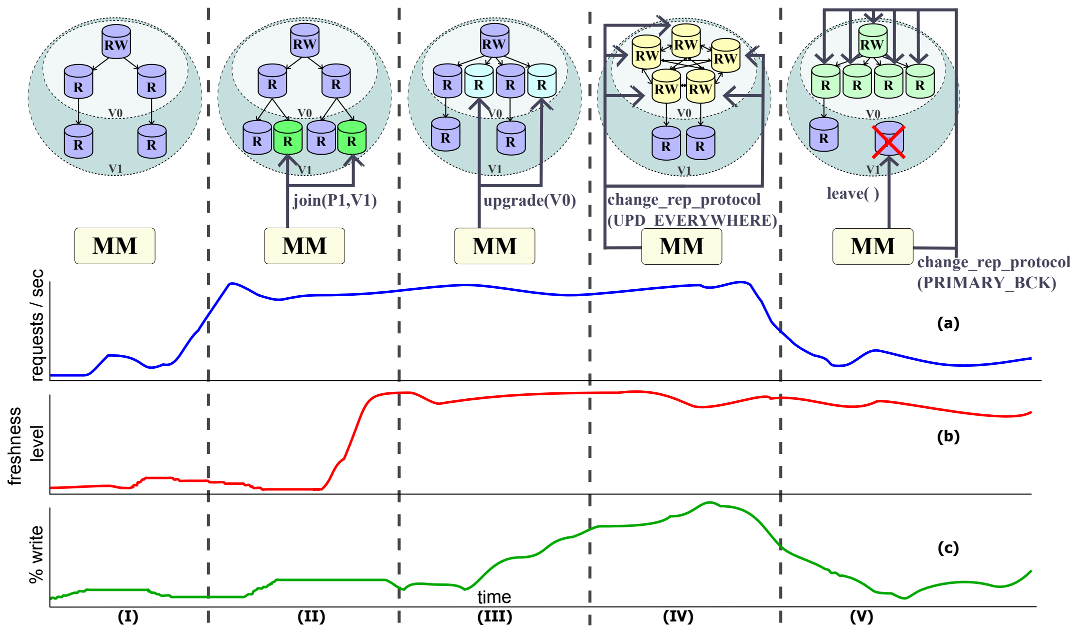


Figure 7.3: Example of interaction between the metadata manager (MM) and system replicas.

second submitted to the system, (b) average freshness level demanded by transactions (where the lower the freshness level is, the more tolerant to accept outdated values the transaction is) and (c) rate of transactions including write operations.

The initial configuration (Scenario I) of the depicted partition consists of three replicas managed by a primary-backup replication protocol in the core hierarchy layer (v0), along with two more replicas in the lower hierarchy layer (v1) that serve as backups for two replicas of the upper layer. Hence, the only replica that can execute update transactions is the primary of the core layer.

An elastic service should dynamically add the necessary resources while interfering with other running processes as little as possible upon detecting a workload increase. In Figure 7.3, when the metadata manager detects a relevant increment in the number of requests per second, it adds two more replicas (Scenario II). Since the number of write operations is low and most transactions do not require a high freshness level, the new replicas are incorporated to the lower hierarchy layer. Consequently, the primary replica will not be burdened by the addition of these new nodes.

On the other hand, it should be taken into account that the freshness level demanded by transactions has an influence on the number of transactions that must be submitted to replicas of the core level, because transactions that require a high freshness level cannot be delegated to lower levels of the hierarchy. Thus, upon detecting an increase in the average freshness level, the metadata manager may need to add more resources to the core level to avoid an overload situation, as in the case of Scenario III.

As the performance of replication protocols varies greatly depending on the read-/write of transactions, the proposed system is also capable of adapting itself to such changes. In Figure 7.3, upon detecting that the number of writes increases, the metadata manager changes the replication protocol of the core layer from a primary-backup scheme to an update-everywhere scheme (see Scenario IV), so that all replicas in the core layer can execute update transactions.

Finally, it should be noted that that apart from adapting itself to tolerate increasing workloads, an elastic service should also minimize resources usage (e.g., by releasing replicas when the workload decreases). This situation is represented in Scenario V, in which the metadata manager removes one of the replicas upon detecting an important decrease in the number of requests submitted to the system. Moreover, since the rate of transactions that include write operations also decreases, the metadata manager changes the replication protocol to primary-backup again.

The previous example concludes the preliminary introduction of our approach. The following section focuses on the details of Epidemia's formal specification concerning the communications network, the clients, and the metadata manager.

7.5 FORMAL SPECIFICATION

Epidemia is composed by a total set of processes $P = C \cup M \cup R$, where C is the set of all client processes, M is the set of metadata manager processes, and R is the set of all possible replica processes. All these processes communicate among themselves by exchanging messages, being $\mathcal{M}$ the set of all possible messages.

Data items belonging to the database stored in Epidemia are distributed among a set $\mathcal{P}$ of disjoint partitions, where $p_k.items$ denotes the set of data items stored at partition $p_k \in \mathcal{P}$. Each partition p_k is stored and managed by a different replication cluster managed by a hierarchy h_k of system replicas, where each h_k comprises a set of replicas and their hierarchical relationships.

A transaction $t_{ij} \in \mathcal{T}$ (being $\mathcal{T}$ the set of all possible transactions) is identified by the pair formed by the identifier of the client $c_i \in C$ that submits the transaction and an increasing local sequence number j (that is incremented for each new transaction that c_i submits).

Each transaction t_{ij} contains the set of operations $t_{ij}.operations = \langle op_1, op_2, \ldots, op_m \rangle$ that must be executed. We denote by $t_{ij}.items$ the set of data items that are accessed or altered by t_{ij}. Moreover, the freshness degree demanded by the transaction is represented in a generic way by $t_{ij}.freshness$, where the higher the value of $t_{ij}.freshness$, the more recent the versions of the retrieved data items must be (recall that the freshness level can be expressed in terms of absolute values, version numbers, or timestamps). Finally, a boolean parameter, named $t_{ij}.isReadOnly$, is used to determine whether t_{ij} is a read-only transaction (if $t_{ij}.isReadOnly = true$) or an update one (if $t_{ij}.isRead\text{-}Only = false$).

7.5.1 *Properties of point-to-point communications*

We assume asynchronous FIFO quasi-reliable point-to-point channels modeled through two primitives as defined in [Schiper, 2006]: $PTPsend(m, q)$ and $PTPdeliver(m, q)$. $PTPsend(m, q)$ is invoked to send a message $m \in \mathcal{M}$ to a process q, whereas $PTPdeliver$ (m, q) is an upcall executed upon the receipt of a message m sent by process q. Also, we assume each message to be tagged with a unique message identifier that distinguishes it from every other message. The guarantees offered by the point-to-point channels (see Chapter 2) used in Epidemia (note that, as point-to-point channels are used by clients, metadata manager nodes and replicas, p and q belong to $P = C \cup M \cup R$) are defined in what follows:

Algorithm 7.1 Execution flow of client c_i.

```
 1: Initialization
 2:    $j \leftarrow 0$  % Local counter to generate unique transaction IDs
 3:    $\mathtt{mmCatalog} \leftarrow (\mathtt{mm}_1, \ldots, \mathtt{mm}_m) \in M$  % Catalog with metadata manager nodes addresses
 4:    $\mathtt{replicaCache} \leftarrow \emptyset$  % Cache for storing the replicas that can execute different transaction types
 5: issueRequest($\langle \mathbf{op_1}, \mathbf{op_2}, \ldots, \mathbf{op_m} \rangle$) : Result
 6:    $j \leftarrow j + 1$
 7:    $t_{ij}.\mathtt{operations} \leftarrow \langle \mathtt{op}_1, \mathtt{op}_2, \ldots, \mathtt{op}_m \rangle$
 8:    Determine $t_{ij}.\mathtt{items}$, $t_{ij}.\mathtt{freshness}$ and $t_{ij}.\mathtt{isReadOnly}$
 9:    Determine $t_{ij}.\mathtt{type}$
10:    do
11:       if $(t_{ij}.\mathtt{type}, \{(p_1, r_1), \ldots, (p_n, r_n)\}) \in \mathtt{replicaCache}$ then
12:          $\mathtt{replicas} \leftarrow \{(p_1, r_1), \ldots, (p_n, r_n)\}$
13:       else
14:          do
15:             Select $\mathtt{mm}_l \in \mathtt{mmCatalog}$
16:             $\mathtt{PTPsend}(\mathtt{MMRequest}\langle t_{ij} \rangle, \mathtt{mm}_l)$
17:             Wait until $(\mathtt{PTPdeliver}(\mathtt{MMReply}\langle t_{ij}, \{(p_1, r_1), \ldots, (p_n, r_n)\}\rangle, \mathtt{mm}_l)$ or
                   timeout expires)
18:             $\mathtt{replicas} \leftarrow \{(p_1, r_1), \ldots, (p_n, r_n)\}$
19:          until $(\mathtt{replicas} \neq \emptyset)$
20:       end if
21:       for $(p_k, r_k) \in \mathtt{replicas}$ do
22:          $\mathtt{PTPsend}(\mathtt{TransactionRequest}\langle p_k, t_{ij} \rangle, r_k)$
23:       end for
24:       Wait until $(\mathtt{PTPdeliver}(\mathtt{TransactionResult}\langle \mathtt{result}_{ij}^k \rangle, r_k)\ \forall k \in \{1, \ldots, n\}$ or time
             out expires)
25:       if $(\exists k \in \{1, \ldots, n\} : \mathtt{result}_{ij}^k = \mathtt{null})$ then
26:          $\mathtt{replicaCache} \leftarrow \mathtt{replicaCache} \setminus \{(t_{ij}, \mathtt{replicas})\}$
27:       end if
28:    until$(\forall k \in \{1, \ldots, n\} : \mathtt{result}_{ij}^k \neq \mathtt{null})$
29:    $\mathtt{replicaCache} \leftarrow \mathtt{replicaCache} \cup \{(t_{ij}, \mathtt{replicas})\}$
30:    $\mathtt{result}_{ij} = \cup_{k=1}^{n} \mathtt{result}_{ij}^k$
31:    return $\mathtt{result}_{ij}$
```

[PTP1] CHANNEL VALIDITY. If a process q receives a message m, then m has been sent by some process.

[PTP2] CHANNEL NONDUPLICATION. Process q receives message m at most once.

[PTP3] CHANNEL TERMINATION. If a process p sends a message m to another process q, and both p and q are correct, then q eventually delivers m.

[PTP4] FIFO ORDER. If some process p executes $\mathtt{PTPsend}(m_1, q)$ before executing $\mathtt{PTPsend}(m_2, q)$ and q executes $\mathtt{PTPdeliver}(m_2, p)$, then q must have executed $\mathtt{PTPdeliver}(m_1, p)$ previously.

Note that PTP4 has been added to the definition of [Schiper, 2006], so as to ensure that point-to-point channels are FIFO and, thus, messages are not reordered with respect to the order in which they were sent.

7.5.2 *Client specification*

A client $c_i \in C$ is a process that submits transactions to the system in order to execute them and obtain the corresponding result. Clients communicate with both metadata manager nodes and replicas using point-to-point channels as defined in Section 7.5.1.

Algorithm 7.1 shows the pseudo-code of the client side. Note that clients must know a subset of metadata manager nodes $mmCatalog = (mm_1, mm_2, \ldots mm_m) \subseteq M$ and be able to access the process running at each alive node through point-to-point channels. Also, we have included a cache (named *replicaCache*) that matches the most used transaction types to the replicas that can execute them, which alleviates the number of queries issued against the metadata manager. This cache consists of a set of pairs of the form $(t.type, \{(p_1, r_1), \ldots, (p_n, r_n)\})$; where each pair indicates that, for a certain transaction type $t.type$, there are n partitions that contain items accessed in transactions of that type, and each $r_k \in R$ (being $k \in \{1, \ldots, n\}$) is a replica that (according to the information provided by the metadata manager) belongs to partition $p_k \in \mathcal{P}$ and can execute transactions of that type. Note that, for single-partition transactions, $n = 1$, whereas for multi-partition transactions, $n > 1$. The type of a transaction is determined depending on the following three parameters: the set of items that it accesses ($t_{ij}.items$), the freshness degree required for read operations ($t_{ij}.freshness$), and whether it is a read-only or an update transaction ($t_{ij}.isReadOnly$).

To execute a transaction in the system, client c_i must use the function $issueRequest$ (see line 5 in Algorithm 7.1). We assume that a client c_i is blocked while executing this function, so that c_i cannot issue two transactions simultaneously (recall that interactive transactions are not supported).

Function $issueRequest$ creates a transaction request t_{ij} identified by the pair formed by the client identifier i and the sequence number j (incremented for every new request), containing the operations to be executed (lines 6-7). Once the transaction type has been determined (line 9), the client checks whether $replicaCache$ contains information about the replica (or replicas in the case of a multi-partition transaction) that can handle the corresponding transaction type (line 11). In this case, the transaction will be sent to the replicas indicated by $replicaCache$. Otherwise, the transaction will be submitted to one of the metadata manager nodes within a $MMRequest$ message, which will (1) determine the partitions involved in the execution of t_{ij}, (2) assign the replicas that should execute it, and (3) send this information to the client in a message of type $MMReply$ (lines 15-18). Note that the loop of lines 14-19 allows the client to cope with failures of metadata manager nodes.

Once the replica (or replicas) that must execute the transaction are known, a transaction request (*TransactionRequest*) containing t_{ij} is submitted to them (lines 21-22). Each replica r_k (where $k \in \{1, \ldots, n\}$, being n the number of partitions involved in the transaction) that received t_{ij} will execute the transaction and send the result $TransactionResult\langle result^k_{ij}\rangle$ back to the client. The identifier of the partition (p_k) that each replica is supposed to belong to is sent to the corresponding replica r_k along with t_{ij}, as r_k must check that it still belongs to partition p_k upon executing t_{ij}.

Algorithm 7.2 Input events of the workload manager.

```
1:  PTPdeliver(MMRequest⟨$t_{ij}$⟩, $c_i$)
2:     $replicas \leftarrow \emptyset$
3:     for each $p_k \in \mathcal{P}$ such that $p_k.items \cap t_{ij}.items \neq \emptyset$
4:        $r_k \leftarrow chooseReplica(t_{ij}, h_k)$
5:        $replicas \leftarrow replicas \cup (p_k, r_k)$
6:     end for
7:     $PTPsend(MMReply\langle t_{ij}, replicas\rangle, c_i)$
8:  PTPdeliver(Heartbeat⟨replicaInfo⟩, $r_k$)
9:     Process $replicaInfo$
10: PTPdeliver(Conf_ACK⟨⟩, $r_k$)
11:    Record ACK
```

Note that the client waits for the results until a timeout expires; if the latter happens without having received all the results, a new request will have to be sent to the metadata manager, in order to find a new valid set of replicas that can execute the transaction. To do so, the set of replicas that failed to execute t_{ij} is removed from the cache (so they will no longer be the predetermined option for executing transactions of type $t_{ij}.type$, line 26), and the external loop comprising lines 10-28 is re-executed until all results are received.

Once the results from the n replicas have been collected, the identifiers of the replica (or replicas) that executed it are stored in the cache, so that subsequent transactions of the same type will be directly sent to them (line 29). Additionally, the received results must be merged in case $n > 1$ to form the final result $result_{ij}$ that is returned by the function (lines 30-31).

For the sake of exposition, we have not included in the pseudo-code of Algorithm 7.1 the task of refreshing $replicaCache$—that consists in periodically querying the metadata manager to obtain the most adequate replicas to execute the most popular transaction types.

7.5.3 *Metadata manager specification*

In the following, we specify the behavior of the different components that are included in the metadata manager: workload manager, replication manager, and transaction manager (see Figure 7.2). For the sake of simplicity, in the following specification we assume that it is a centralized process—despite it should be replicated among a small set of nodes in order to avoid becoming a bottleneck and single point of failure (alternatively, clients could hold a read-only copy of the metadata, as done in Omid [Gómez Ferro et al., 2014], in order to minimize the dependency of the metadata manager).

Workload manager. The workload manager is in charge of monitoring the behavior of active replicas and handling request messages from clients (so as to determine the replicas that will execute the requested transactions). Therefore, it must balance the load associated to system replicas in order to save resources and establish the optimal system configuration depending on the workload characteristics at each time.

The possible input events that the workload manager can receive are represented in Algorithm 7.2: request messages coming from clients (lines 1-7), monitoring information received from replicas via heartbeat messages (lines 8-9), and acknowledgement messages that replicas send in response for configuration messages determined by the metadata manager (lines 10-11). The actions resulting from these input events are detailed in what follows.

On the one hand, clients query the workload manager in order to know the replicas that can execute the transactions they want to submit to Epidemia (see Section 7.5.2). As shown in Algorithm 7.2, upon receiving a request $MMRequest$ for a transaction t_{ij} (line 1), the workload manager must determine all the partitions involved in the execution of t_{ij}, that is, all partitions that contain at least one item that is accessed or modified by t_{ij} (line 3). For each of the involved partitions p_k, it chooses one replica of the hierarchy h_k that handles p_k. This election, represented by function $chooseReplica$ (line 4), must take into account the requirements of t_{ij} (i.e., the demanded freshness degree and whether it is a read-only or an update transaction) and include load balancing mechanisms to distribute client requests among the replicas of the hierarchy depending on the current workload of replicas. Thus, the implementation of this function requires complex decision-making algorithms [Curino et al., 2010] that are out of the scope of this work. Once the replica in charge of executing the transaction at each involved partition is selected, the workload manager sends an $MMreply$ message back (line 7) to the client including the address (or addresses) of the selected replica (or replicas).

On the other hand, heartbeat messages that are periodically received from replicas serve to monitor their status (lines 8-9).

Using the information collected from heartbeat messages, the workload manager is able to track the status of the replication clusters and, thus, make decisions on the optimal system configuration depending on the workload characteristics and resources behavior. Hence, the workload manager must include a set of rules to dynamically adapt the system configuration to the current workload. Note that the specification of an intelligent decision-making tool that defines these rules is out of the scope of this work [Curino et al., 2010; Sancho-Asensio et al., 2014; Porobic et al., 2014].

Upon deciding that a certain change in the configuration must be performed, the workload manager will send a message to the involved replicas, so as to reconfigure them in the appropriate manner. The different configuration messages (labeled under the generic message type $Conf\_Msg$) that the workload manager can send to a replica r_n are the following:

- $Join\langle p_k, h_k, level, r_{rec}\rangle$: this message will be sent to a replica r_n in order to make it join the hierarchy h_k (which manages partition p_k) at level $level$. Thus, r_n will have to retrieve the current system state from replica r_{rec}. Since we are considering in-memory databases, when a replica joins a replication cluster, it will obtain all the data items necessary to belong to the level it joins to. In case r_n joins the highest level of the hierarchy, r_{rec} will have to be one of the replicas that already belonged to the core level. Otherwise, r_{rec} will be r_n's parent replica.
- $Leave$: it tells a replica to leave the system and become offline.

- Upgrade(level,r_p): when a replica r_n has to be upgraded to a hierarchy level with a more recent version, this message is sent to r_n to inform it of its new location and to tell it that its new parent will be replica r_p. In order to become a member of this level, r_n will have to retrieve from r_p the necessary data items to become updated.
- Downgrade(level,r_p): when the metadata manager wants to downgrade a replica r_n to a hierarchy level with a staler version of data items, it sends a message of this type, informing r_n of its new location in the hierarchy. From then on, r_p will be r_n's parent in the hierarchy. Note that, since r_n belonged to a higher level in the hierarchy, it may receive some updates from its parent that r_n may have already applied before being downgraded, so it will have to detect these updates and discard them.
- Add_Child(r_c): this message is sent to a replica r_n to inform that, from then on, r_n will have to asynchronously propagate all the updates it applies to replica r_c.
- Remove_Child(r_c): this message is sent to inform r_n that it will no longer have to propagate its updates to replica r_c.

Note that other configuration actions can be performed by combining some of these messages. For instance, a replica can be moved from a replication cluster to another one by combining a Leave message—to force the replica quitting from its current replication cluster—with a Join message—to make it join another replication cluster.

Once r_n has received and applied a configuration message, it will send an acknowledgment message back to the metadata manager, which will record that r_n has actually updated its configuration (lines 10-11).

In addition to the aforementioned Conf_Msg messages, the workload manager can send Ping messages to check whether a replica that has not sent its Heartbeat messages for a while is still alive or can be considered as crashed. When a replica receives a Ping message, it should answer with a Heartbeat message. If a timeout expires without having received a reply from the replica, the workload manager will assume that the replica has crashed; hence, it will remove it from the hierarchy and will no longer forward client requests to it.

Finally, recall that the workload manager is also in charge of determining the partitioning scheme and adapting it to the current demands by dynamically splitting and merging partitions. This requires to find the optimal strategy to minimize multi-partition transactions while making the best possible use of available resources by means of a partitioning algorithm [Cheung et al., 2012; Curino et al., 2010; Pavlo et al., 2012; Trushkowsky et al., 2011; Sancho-Asensio et al., 2014; Porobic et al., 2014].

Transaction manager The transaction manager is responsible for orchestrating the correct execution of multi-partition transactions. As discussed in Section 7.2.2, there are several ways for coordinating transactions that access different partitions, in order to globally maintain consistency [Curino et al., 2010; Pandis et al., 2011a]. In a nutshell, the transaction manager designates at least one replica from each involved partition to lead the execution of the transaction in its partition and contact other replicas of other involved partitions to synchronize the execution of the transaction if necessary.

Replication manager The replication manager chooses the replication protocol that best fits at the core level of each hierarchy depending on the available resources and the workload characteristics, making use of the information collected by the workload manager.

Upon detecting the need for changing the replication protocol to improve the efficiency of a hierarchy of replicas, the replication manager will send a configuration message (Conf_Msg) named Change_Replication_Protocol to the replicas involved in the change. The replication manager also ensures that the transition from the old replication protocol to the new one does not compromise data consistency.

7.5.4 *Replica specification*

In order to define the status of a replica r_n, let us consider the following variables:

- p. Partition managed by the replication cluster that r_n belongs to, which is the partition specified in the Join message by which r_n is ordered to join the system.
- level. Hierarchy level in which r_n is located, where the lower the hierarchy level is, the closer to the core level it is located. The hierarchy level is initially set in the Join message by which r_n is ordered to join the system, and can be changed by means of Upgrade or Downgrade messages. The level of the replica is only set once a replica is updated with regard to the hierarchy level it belongs to; i.e., once the updates derived from applying the corresponding Join or Upgrade messages have been applied (we can assume that level takes a null value before that). Consequently, r_n cannot process client requests (as shown in the following subsection) until it contains the adequate versions of data items.
- freshness. Freshness degree provided by the hierarchy level in which r_n is located, where the higher the freshness degree the more recent the versions of stored data items are. Thus, replicas located in hierarchy levels that are close to the core level will have a higher freshness degree than replicas located in lower hierarchy levels.
- replicationProtocol. In case r_n belongs to the core level of a hierarchy, this variable represents the instance of the replication protocol running at r_n.
- isReadOnly. Boolean value that states whether r_n can solely execute read-only transactions (in case this variable is set to true), or can also execute update transactions (if it is set to false). More specifically, this variable will be set to false only in replicas belonging to the core level that can execute update transactions. For instance, if the core level of a replication cluster is managed by an update-everywhere replication protocol, this variable will be false for all replicas belonging to the core level; in contrast, in case a core hierarchy level is handled by a primary-backup protocol, isReadOnly will be false only for the primary replica.

Algorithm 7.3 Input event representing the delivery of a transaction request to replica r_n.

```
1:  PTPdeliver(TransactionRequest⟨$p_k$, $t_{ij}$⟩, $c_i$)
2:    if $p = p_k$ then
3:      if $t_{ij}$.isReadOnly and freshness $\geqslant t_{ij}$.freshness then
4:        $result^n_{ij} \leftarrow$ execute($t_{ij}$)
5:        PTPsend(TransactionResult⟨$result^n_{ij}$⟩, $c_i$)
6:      else if (not $t_{ij}$.isReadOnly) and ($level = 0$) and (not $r_n$.isReadOnly) then
7:        replicationProtocol.process($t_{ij}$)
8:        % else discard transaction
9:      end if
10:     % else discard transaction
11:   end if
```

- children. List containing r_n's children replicas (i.e., the replicas belonging to the next hierarchy level to which r_n must asynchronously propagate its updates).

By making use of these variables, the remainder of this section specifies how (1) system replicas handle incoming transactions, (2) replicas of the core level of each hierarchy make use of the guarantees provided by GCSs to propagate updates among themselves by means of a replication protocol, (3) updates are propagated among hierarchy levels, and (4) replicas communicate with the metadata manager.

Processing client requests. Algorithm 7.3 describes the actions that happen at a replica r_n upon receiving a client request containing a transaction t_{ij} to be executed. Although it has not been included in Algorithm 7.3 for the sake of simplicity, in case the replica has to discard t_{ij} (lines 8 or 10) for any of the reasons taken into account in this algorithm, it would be convenient to send an error message to the client so that it can ask the metadata manager for a replica able to handle t_{ij}.

First, r_n checks that the partition where the client intends to execute t_{ij} is the same as the one managed by the replication cluster that r_n belongs to (line 2). Otherwise, the transaction must be discarded.

In case that t_{ij} is a read-only transaction, the freshness related to the hierarchy level to which r_n belongs must be capable of fulfilling the freshness limit demanded by t_{ij} (line 3). If this statement is satisfied, the replica executes t_{ij} and sends the result to the client (lines 4-5). Here we are assuming that read-only transactions are executed without delay, although the execute function could introduce a delay in the execution of the transaction in order to meet different consistency constraints by waiting for the replicas to converge to a certain state, thus adding complexity to the notion of freshness. For instance, in case a transaction demanded read-your-writes consistency [Vogels, 2009], the replica would have to ensure that all update transactions t_{ik} such that k j (i.e., all update transactions from client c_i that have been issued before t_{ij}) have been applied at r_n before executing t_{ij}.

On the other hand, if t_{ij} is an update transaction (i.e., $t_{ij}.isReadOnly = false$), it can only be processed by r_n if this replica belongs to the core level of the hierarchy that manages p_k (i.e., $level = 0$) and is not a read-only replica (i.e., $isReadOnly = false$). In case this is true, the transaction is delegated to the replication protocol, which will be in charge of executing and propagating the changes performed by t_{ij} (line 6). Otherwise, t_{ij} cannot be executed at r_n.

Before detailing the steps that the replication protocol follows in order to ensure the correct execution of updates, we will describe the properties provided by the GCS, which is used by the replication protocol in order to broadcast messages to the replicas of each core level.

Properties of GCSs. With the aim of formalizing the communication guarantees provided by the GCSs that encompass the replicas of each core level in Epidemia, we have followed the specifications for dynamic multicast given in [Schiper, 2006].

More specifically, we consider a view-oriented GCS (see Chapter 2) that informs of membership changes by means of view changes. A *view* is a tuple $v = (i, S)$, where i is an integer that denotes the identifier of view v, and S is a non-empty subset of R that represents the membership of v. For the sake of simplicity, a process p is said to be in view v (i.e., $p \in v$) if $p \in v.S$.

In a dynamic GCS (in which participating processes can be added or removed during the computation, as in the case of Epidemia), requirements must be put on processes that are members of the group but do not crash, because faulty processes and correct processes that do not belong to the group have no obligations with respect to the messages multicast to the group. We formalize this in the context of a given view v of some group g by introducing the notions of $v\_correct$ process and $g\_correct$ process as defined in [Schiper, 2006]:

1. A process p is said to be $v\_correct$ if (1) p installs view v, with $p \in v$; (2) p does not crash while its view is v; and, (3) if v is not the last view of some process in v, then there exists view v' installed immediately after v by some process in v such that $p \in v'$.
2. Let v_{init} be the initial view of process p for group g, and $p \in v_{init}$. Process p is said to be $g\_correct$ if (1) p is $v_{init}\_correct$ and (2) there exists no view v' such that p is in v' and p is not $v'\_correct$.

Each replica process p has access to two primitives that define uniform reliable multicast, namely $GCSmulticast(m)$ and $GCSdeliver(m)$, used for multicasting and delivering a message m respectively. Uniform reliable multicast is defined by the following properties.

CLAIM 5. **Validity.** If a $g\_correct$ process executes *GCSmulticast(m)*, then it eventually executes $GCSdeliver(m)$.

CLAIM 6. **Uniform agreement.** If a process executes $GCSdeliver(m)$ in view v, then all processes that are $v\_correct$ eventually execute $GCSdeliver(m)$.

CLAIM 7. **Uniform integrity.** For any message m, every process executes $GCSdeliver(m)$ at most once and only if a process executed $GCSmulticast(m)$.

CLAIM 8. **Uniform same view delivery.** If two processes p and q execute $GCSdeliver(m)$ in views v and w respectively, then $v = w$.

Note that the combination of the $Claim\ 5$ and $Claim\ 6$ properties ensures virtual synchrony (see Chapter 2) as proofed by Schiper [2006].

Algorithm 7.4 Input event for the delivery of an asynchronous update to replica r_n from its parent replica r_p.

```
1: PTPdeliver(TransactionWriteset⟨$w_{ij}$⟩, $r_p$)
2:   apply($w_{ij}$)
3:   commit
4:   for each $r_c \in$ children do
5:     PTPsend(TransactionWriteset⟨$w_{ij}$⟩, $r_c$)
6:   end for
```

Nonetheless, the above specification does not take into account any ordering guarantees apart from those provided by Claim 8, which translates into the fact that messages delivered between two consecutive view changes can be delivered in any order, and that order does not have to be the same at all processes that install the two view changes. In order to establish an order in the sequence of multicast messages, we make use of two ordering guarantees: (1) FIFO order, which ensures that messages multicast by a given process are delivered according to the order in which the process sent them (i.e., FIFO uniform reliable multicast) ; and total order, which guarantees that all view members deliver messages in the same order, irrespective of which process multicast them (i.e., total order uniform reliable multicast).

Therefore, the properties of FIFO uniform reliable multicast comprise Claims $5-8$ from uniform reliable multicast, in addition to the following constraint:

Claim 9a. **Uniform FIFO order.** If some process executes GCSmulticast(m_1) before it executes GCSmulticast(m_2), and some process p executes GCS-deliver(m_1) in view v, then every process in view v (including p) executes GCSdeliver(m_2) only after it has executed GCSdeliver(m_1).

Also, the properties of total order uniform reliable multicast includes properties Claims $5-8$ of uniform reliable multicast, plus an ordering property:

Claim 9b. **Uniform total order.** If some process executes GCSdeliver(m_1) in view v before it executes GCSdeliver(m_2), then every process p in view v executes GCSdeliver(m_2) only after it has executed GCSdeliver(m_1).

It is worth noting that these two last properties forbid gaps in the delivery sequence of messages, and therefore avoid the problem of contamination [Défago et al., 2004].

Additionally, the GCSs that manage the replicas of the core level of each hierarchy in Epidemia provide the group membership guarantees specified in [Schiper, 2006], where the aforementioned properties of uniform reliable multicast are used for defining the execution order of join and leave operations; plus an additional initialization property that defines the initial view of process p to be either the initial view of the group or a view installed by some other process q.

Therefore, according to the specifications of [Schiper, 2006], the join and leave operations are the only means to modify the membership and produce a view change in the GCSs of Epidemia. Process p requests to add process q to the group by invoking the operation join(q), but the view only changes when the

Algorithm 7.5 Messages delivered from the workload manager at replica r_n.

```
1: PTPdeliver(Conf_Msg⟨params⟩, mm_l)
2:    Apply configuration message using params
3:    PTPsend(Conf_ACK⟨⟩, mm_l)
4: PTPdeliver(Ping⟨⟩, mm_l)
5:    replicaInfo ← Collect replica status
6:    PTPsend(Heartbeat⟨replicaInfo⟩, mm_l)
7: When heartbeat_timeout expires
8:    replicaInfo ← Collect replica status
9:    PTPsend(Heartbeat⟨replicaInfo⟩, mm_a)
```

$join(q)$ operation is scheduled for execution. Similarly, p requests to remove q from the group by invoking the operation $leave(q)$, and the view changes once the operation is scheduled for execution. The invocation of operations $join$ and $leave$ is denoted by $join\_inv$ and $leave\_inv$ respectively. The execution of $join$ and $leave$ is denoted by $join\_exec$ and $leave\_exec$. Moreover, operation $init(v)$ is used for initializing the view of a process.

Replication protocols. Every database replication protocol that runs at the core level of each hierarchy must provide the following functionalities:

1. An implementation of function $process(t_{ij})$, which is invoked upon receiving an update transaction processed by r_n (see Algorithm 7.3). This function must ensure that updates are correctly propagated to all the replicas belonging to the same core level as r_n.
2. Duplicate requests must be identified, so as not to execute them twice.
3. Updates must be propagated to the replica's children.

For the sake of this dissertation, we propose to develop two different database replication protocols: a primary-backup protocol and an update-everywhere protocol (see Chapter 2 and Chapter 3) whose formal specifications can be found in Bartoli [1999].

Propagation of updates among hierarchy levels. When an update from the parent replica r_p is received at replica r_n, it must apply the update and then also propagate it to its own children replicas by sending messages through the point-to-point connections, as described in Algorithm 7.4. We assume that the updates of a transaction t_{ij} are propagated from r_n to the rest of replicas of the core level and to its children replicas in the form of a writeset ws_{ij}, instead of sending the whole SQL statement and parameters. Thus, when a replica receives a writeset from its parent, it can directly apply it instead of executing the transaction.

Interaction with the metadata manager. Algorithm 7.5 shows the events that may be triggered at a replica by the delivery of a message from the metadata manager.

Upon receiving a configuration message from the workload manager (see Section 7.5.3), the replica applies the configuration change determined by this message, and answer the workload manager with an acknowledgement message (see lines

1-3 of Algorithm 7.5). Recall that $\mathrm{Conf\_Msg}$ is the generic message type that comprises all configuration messages that can be generated by the different components of the metadata manager. On the other hand, when either a timeout expires (lines 4-6) or the replica receives a Ping message from a metadata manager node (lines 7-9), the replica sends a $\mathrm{Heartbeat}$ message to a metadata manager node containing information about its status.

After finishing the formal specification of the system and prior evaluating its formal correctness, an analytical model to compute the scale out factor of Epidemia is proposed.

7.6 ANALYTICAL MODEL

So far, a very first attempt to estimate the scale out factor of a distributed database has been proposed in [Serrano et al., 2007] and used in Chapter 5. We present an alternative approach of this model to make it appropriate for large-scale multi-partitioned and multi tenant distributed databases, thus making it suitable for the architecture herein proposed.

7.6.1 *Scale out formula*

Recall that the scale out factor ($\mathrm{ScaleOut}$) computes the performance of a replicated database versus a non-replicated one [Serrano et al., 2007]. Assuming that (1) every site_i has a processing capacity C (i.e., amount of transactions per second that it can process) and (2) each site in the replicated database performs an amount of local work L_i, the scale out is computed as follows:

$$\mathrm{ScaleOut} = \frac{\sum^{\forall \mathrm{site}_i} L_i}{C}. \tag{7.1}$$

Given that—ideally—the amount of local performed work at site i should be equal to C ($L_i = C$), the scale out factor of a fully replicated database with N sites should increase linearly according to Equation 7.2.

$$\mathrm{ScaleOut}_{\mathrm{ideal}} = \frac{\sum_{i=1}^{N} C}{C} = N. \tag{7.2}$$

This makes sense because—ideally—data should be N times more available in the replicated database. However, Equation 7.2 is not considering the cost of applying the replication protocol over the database (also referred to as replication process overhead). Hence, the total amount of work performed at site i (work_i) is grained as:

$$\mathrm{work}_i = \mathrm{LocalTransactions}_i + \mathrm{RemoteTransactions}_i. \tag{7.3}$$

Indeed, the amount of local work at site i (L_i) is the sum of (a) the work arisen from the local read and/or update transactions issued against the items stored at site i (*LocalTransactions*$_i$) plus (b) the work arisen from the remote update transactions issued by the replication protocol and performed against the items stored at site i ($\mathrm{RemoteTransactions}_i$). Recall that no read transactions will be deferred by the replication protocol. Thus, we model the cost of applying the operations issued against site_i with the following parameters:

- R_i: amount of read-only transactions per second issued against $\mathtt{site}_i$.
- LU_i: amount of local update-only transactions per second issued against $\mathtt{site}_i$.
- RU_i: amount of remote update-only transactions per second issued against $\mathtt{site}_i$.
- w_o: this parameter, ranged between 0 and 1, models the fact that remote updates are cheaper to execute than local updates (they are seen as key/value tuples that do not demand further SQL parsing).
- R_d: number of replicas of each item.
- C_i: amount of transactions per second that $\mathtt{site}_i$ can afford.

Hence, Equation 7.3 is rewritten as follows:

$$work_i = \underbrace{R_i + LU_i}_{\mathtt{LocalTransactions}_i} + \underbrace{w_o \cdot (R_d - 1) \cdot RU_i}_{\mathtt{RemoteTransactions}_i} \leqslant C_i. \tag{7.4}$$

Recall that the scale out factor only considers the amount of local work performed at each site (i.e., read/update local transactions). Assuming that all sites perform at their best, according to Equation 7.4, the amount of local work at $\mathtt{site}_i$ can be expressed as follows:

$$L_i = C_i - w_o \cdot (R_d - 1) \cdot RU_i. \tag{7.5}$$

From Equations 7.1 and 7.5, we obtain the final formula to compute the scale out of a distributed database:

$$\mathtt{ScaleOut} = \frac{1}{C} \sum_{i=1}^{N} (C_i - w_o \cdot (R_d - 1) \cdot RU_i). \tag{7.6}$$

This is a general formula which can be applied in several scenarios if tuned properly. The following subsections discuss some aspects of this model and provide some hints and examples on how to use it in other domains.

7.6.2 *Granularity of transactions*

So far, we have assumed that the system is able to process read only and/or update only transactions. Actually, this situation is not very common in real systems since several read and update operations might be constrained within a single transaction (e.g., read item x and update its value with y):

$$T_0 = \{b_0, r_0(x), r_0(y), w_0(x|y), r_0(y), c_0\}$$

Recall that our model only computes the work carried out by each node. Thus, in terms of work physically performed in a site (i.e., scale out), the aforementioned transaction could be equivalently split into one read only transaction and one update only transaction as follows:

$$T_0 \equiv T_{0r} \cup T_{0w} = \{b_{0r}, r_0(x), r_0(y), r_0(y), c_{0r}\} \cup \{b_{0w}, w_0(x|y), c_{0w}\}$$

Our scale out formula proposal considers neither the cost of begins and commits nor the correctness of the transactions applied to the system; we are rather interested in the different types of operations inside a transaction.

Without generalization loss, our analytical model (Equation 7.6) considers that all transactions have the same number of operations. However, in some situations it might be unfair to consider that transactions with a considerable difference in their number of operations have the same execution cost. In this case, our model is also valid if transactions are once again properly split. For instance, consider the following two transactions (for simplicity, we are assuming read only transactions):

$$T_0 = \{b_0, r_0(x), r_0(y), r_0(z), c_0\},$$
$$T_1 = \{b_1, r_1(m), r_1(n), c_1\}.$$

In order to make them suitable for our analytical model, they should be split into transactions containing the greatest common divisor (g.c.d.) of the number of operations contained in each transaction (i.e., g.c.d.$(3,2) = 1$) as follows:

$$T_0 \equiv T_{0a} \cup T_{0b} \cup T_{0c} = \{b_{0a}, r_0(x), c_{0a}\} \cup \{b_{0b}, r_0(y), c_{0b}\} \cup \{b_{0c}, r_0(z), c_{0c}\},$$
$$T_1 \equiv T_{1a} \cup T_{1b} = \{b_{1a}, r_1(m), c_{1a}\} \cup \{b_{1b}, r_1(n), c_{1b}\}.$$

Once again, splitting transactions has no effect on the overall system scale out; i.e., the amount of working operations remains constant regardless of the number of fragments a transaction is split into. However, recall that when splitting transactions, the total capacity of the system (C_i) has to be set again according to the new transaction length.

In summary, if there are no strong warranties about the fact that: (1) all transactions have the same number of operations; and, (2) read and update operations are not always isolated in different transactions, the following steps must be carried out:

1. Split all transactions into read-only and update-only transactions.
2. Compute the g.c.d. of all transactions' sizes.
3. Split all resulting transactions from step 1 into transactions with the number of operations computed in step 2.
4. Analytically, set the capacity of each site according to the new transaction size.

7.6.3 *Multi tenancy*

Cloud-based systems are known to take the maximum profit from their physical resources and elastically adapt themselves to the current user demands. To achieve this goal, it is very common to run different applications and their associated databases over the same physical hardware, which is known as multi tenancy. Three multi tenancy models have been explored so far [Jacobs and Aulbach, 2007]: shared table, shared process, and shared hardware.

Regardless of the model chosen by the designer, our analytical model is able to compute the scale out factor, as it is focused on the work performed by the physical hardware rather than the application. Indeed, when dealing with a shared table or a shared process model, the usage of the analytical model is quite straightforward: each application running in each $\mathtt{site}_i$ will be assigned a portion of its total capacity (C_i) acknowledging that:

$$C_i|_{\text{Shared table}} = C_i|_{\text{Shared process}} = C_{i_{App_1}} + C_{i_{App_2}} + \ldots + C_{i_{App_M}} \tag{7.7}$$

Although the same idea is used in the shared hardware model, i.e., split the capacity of each site to meet the needs of each application, the previous equation is not true due to the overhead introduced by the virtualization process [Curino et al., 2011a]. Hence, it should be understood as follows:

$$C_i|_{\text{Shared hardware}} \gg C_{i_{App_1}} + C_{i_{App_2}} + \ldots + C_{i_{App_M}} \tag{7.8}$$

7.6.4 *Partitioning and partial replication*

It is well known that network partitioning in a database boosts its scalability but reduces its data availability and/or consistency [Brewer, 2012]. The architecture proposed in this chapter benefits from this statement by keeping as much data availability and consistency in each partition as possible. This should lead to a tunable architecture able to meet different degrees of scale out according to the demands of each application. Indeed, this can also be reflected in our analytical model considering that each $\mathtt{site}_i$ has different loads (e.g., loads in an update-everywhere partition driven by an update-intensive scenario will be radically different than in a primary-backup partition).

In addition, as the model only considers the amount of work carried out by the hardware, partial replication scenarios can be best seen as a particular case of network partitioning mixed with a multi tenant environment. Thus, Equation 7.6 should be updated in order to consider that there are K different degrees of replication for items contained in $\mathtt{site}_i$ as follows:

$$\text{ScaleOut}|_{\text{Partial replication}} = \frac{1}{C}\sum_{i=1}^{N}\left(C_i - \sum_{j=1}^{K}\left(w_o \cdot (R_{dji} - 1) \cdot RU_i\right)\right). \tag{7.9}$$

As shown in Equation 7.9, each $\mathtt{site}_i$ may have different replication policies (j) depending on the item contained in a transaction. Recall that if a transaction contains items with different degrees of replication, then it can be split as previously demonstrated.

7.6.5 *Analytical results*

Finally, we present the results of using this analytical model in the distributed database architecture proposed in this chapter. For the sake of simplicity, we assume that there are two partitions within the database: one using a primary-backup scheme (intended to host web services) and the other one using an update-everywhere scheme (intended to host profile data and logs arisen from the web services). We also assume that each site can process up to 5000 transactions per second, and is submitted to a 150 transactions per second load. For these experiments, the write overhead (w_o) is set to 0.25. We propose three scenarios with different loads: update intensive, read intensive, and 50% read / 50% update. Then we show how assigning more sites to a partition rather than to the other one impacts in terms of overall scale out.

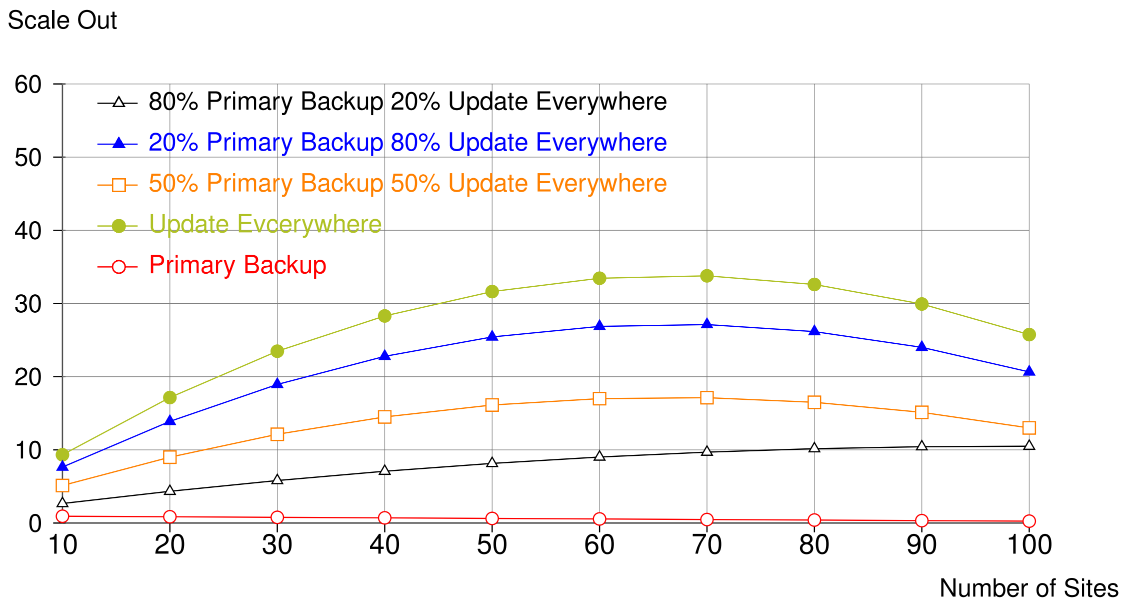


Figure 7.4: Scale out in terms of number of sites in an update-intensive scenario.

7.6.5.1 *Analytical scale out in an update intensive scenario*

Figure 7.4 depicts the scale out behavior of the proposed architecture when each site is submitted to 150 update-only transactions per second (tps). On the one hand, it is shown that a single partition with a primary-backup scheme (empty-circled red line in Figure 7.4) cannot afford this load because there is a single site performing the update operations. Furthermore, the more sites are added to the primary-backup partition, the more deferred updates the master has to send, and thus the less local updates it can process.

On the other hand, a single partition with an update-everywhere scheme (solid-circled yellow line in Figure 7.4) can raise its throughput up to a certain point (60 sites) as all sites in the scenario can run the local update transactions. However, as the number of sites rises, the cost of deferring updates to the rest of replicas is higher and thus the throughput plunges.

Nevertheless, if the system is partitioned as our architecture proposes, the system can reach an arbitrary value of scale out between the upper bound delimited by (1) the update-everywhere single partition and (2) the primary-backup single partition, if the number of sites are assigned to each partition properly. For example, the solid-blue triangled line in Figure 7.4 shows the scale out behavior when assigning the 80% of nodes (in this case 80 nodes) to the update-everywhere partition and the 20% of nodes to the primary-backup partition.

However, it must be taken into account that when the scale out starts decreasing (e.g. beyond 70 sites in the single update-everywhere partition in Figure 7.4), the system is not able to process the input load and a certain amount of transactions are aborted, which obviously derives in a considerable overhead.

7.6.5.2 *Analytical scale out in a read intensive scenario*

This analytical model also provides reliable results when examining a read-intensive scenario (e.g., a web application or a content delivery network). Such environments are

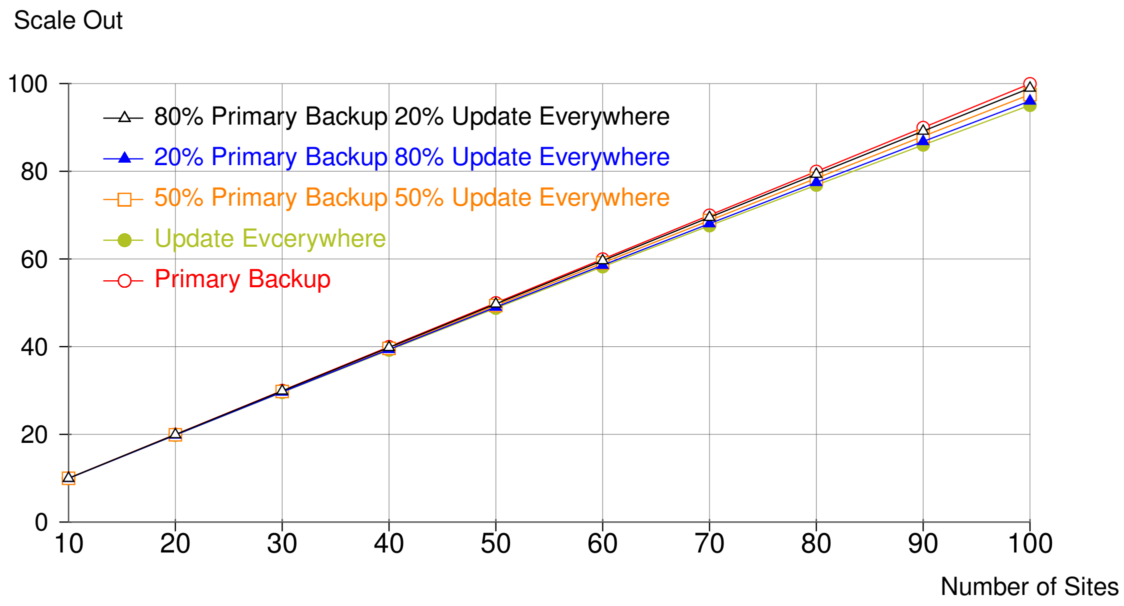


Figure 7.5: Scale out in terms of number of sites in a read-intensive scenario.

aimed at serving multiple read operations once a single write is issued (also referred to as WORM scenarios). In Figure 7.5, we have modeled this situation by submitting each node to a load of both 10 tps update-only tps and 140 read-only tps.

Again, to ease the comparison between different experiments, we have assumed a system able to (1) implement two different replication strategies (update-everywhere and primary-backup) and (2) manage two partitions. Actually, as read operations are no longer propagated to any replicas, both replication strategies should scale-out linearly as long as each site can afford its input load. This is shown in Figure 7.5, where the differences in terms of scale-out between both partitions are minimal. Indeed, the update-everywhere partition (solid-yellow circled line) cannot perfectly scale out linearly because each site has to propagate its 10 tps to the rest of replicas.

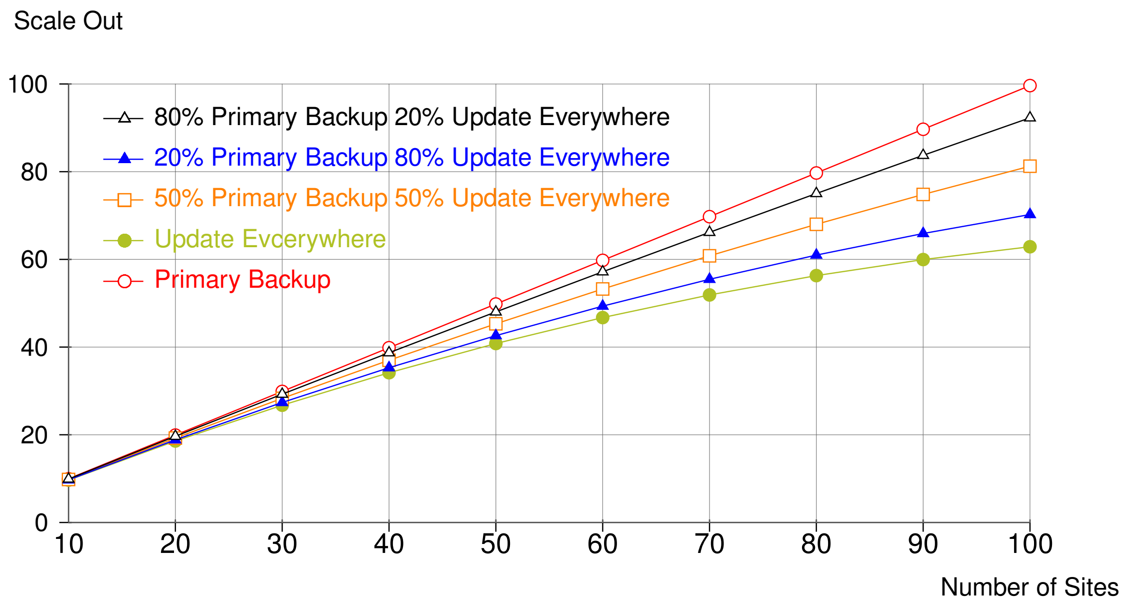


Figure 7.6: Scale out in terms of number of sites in a 50%-read/50%-update scenario.

7.6.5.3 *Analytical scale out in a 50% read / 50% write scenario*

In Figure 7.6 we evaluate a hybrid load scenario where each site is submitted to 75 update-only tps and 75 read-only tps. On the one hand, we can see that in this case the update-everywhere partition scales better than the update intensive scenario as expected; i.e., the less number of updates processed by a site, the less operations have to be deferred and thus the greater scale out is achieved.

On the other hand, we can see in Figure 7.6 that the primary-backup partition scale out behavior is linear, which at first glance might be shocking. As done in Section 7.6.5.1, we have assumed that, along the primary-backup partition, all update-only transactions (intended to be uniformly driven to all sites) are issued against the primary site. This leads to a scenario where the primary master rapidly collapses due to the huge amount of update-only transactions, but the rest of replicas can easily process the incoming read-only transactions. Thus, the scale out keeps rising linearly because the read-only transactions can still be processed without penalizing the global system throughput.

Overall, we have proposed an analytical model to compute the scale out of distributed databases, running under different conditions concerning partitioning and replication protocols. Furthermore, we have applied this analytical model to the distributed storage architecture proposed in this chapter and demonstrated that several degrees of scale out can be obtained if partitions and replication protocols are tuned properly. We have shown that the scale out behavior is directly related with the nature of each application, since different results have been obtained under the same scenario by changing the appliance characteristics (i.e. read and write patterns). The following section develops a correctness proof for Epodemia as a complement for this analytical model.

7.7 CORRECTNESS PROOF

The different Epidemia's components provide certain guarantees that will be taken as a basis point to discuss system correctness in terms of both safety and liveness. We assume that the system does not tolerate Byzantine failures or malicious behavior.

7.7.1 *Guarantees provided by the metadata manager*

As it has been explained in Figure 7.4, the metadata manager is assumed to be distributed among a small set of nodes, which can be synchronized using Paxos [Lamport, 1998] (or an open source variant of Paxos such as Zookeeper [White, 2011]) to provide fault tolerance while guaranteeing consistency. Paxos ensures that the information stored at the metadata repository is not lost nor left in an inconsistent state even in the presence of arbitrary failures, including network partitions. Thanks to Paxos, the following guarantees are ensured at the metadata manager:

- **Safety.** Only a single metadata manager node can own a lease (a *znode* in the case of Zookeeper) at any instant of time, which gives permission to modify the data stored in the metadata repository.
- **Liveness.** The progresses if a majority of nodes are non-faulty and can communicate among themselves.

7.7.2 *Guarantees provided by the replication clusters*

The metadata manager is in charge of determining the set of replicas that constitute the core hierarchy level of each replication cluster ($\mathcal{RC}$). Thus, since at any time there exists a correct metadata manager node that properly controls the behavior of the core replicas (by determining the replication protocol that is executed at the core level, as well as the replicas that must join or leave this hierarchy level), the correctness at the core layer of each $\mathcal{RC}$ depends on the behavior of the replicas that belong to it. Recall that replicas that do not belong to the core level cannot execute update transactions.

The replication protocol running at the $\mathcal{RC}$ core level is devoted to maintain the consistency at this level. We can guarantee that the replication protocol ensures data consistency of the corresponding partition as long as it has been shown to be correct [Daudjee and Salem, 2006] and provides one copy schedules (see Chapter 2) even in the presence of failures [Bernstein et al., 1987; Fekete and Ramamritham, 2010]. Note that, although the metadata manager is responsible for deciding the replicas that belong to the core level of each $\mathcal{RC}$, a replica can only be considered as part of the core level once it has joined the GCS managing that $\mathcal{RC}$ and is therefore included in the group. The virtual synchrony property provided by the GCS (see properties $\mathtt{Claims}\ 5-8$ from uniform multicast in Section 7.5.3) ensures that messages that are multicast to the group are ordered with respect to view changes; hence, all replicas that belong to two consecutive views receive the same set of multicast messages.

Therefore, if a transaction accesses only the core nodes of a single partition, it will behave as if it were executed in a traditional replicated database; hence, the consistency criterion fulfilled will correspond with the consistency guarantees ensured by the replication protocol that manages that core level, normally 1SR or 1CSI depending on the replication protocol. In case a single partition transaction accesses other levels of the hierarchy apart from the core level, the consistency criterion fulfilled will be 1CSI, as update transactions are serially executed at the replicas of the core level whereas read-only transactions can be forwarded to lower hierarchy levels assuming that they might obtain a stale (but consistent) snapshot of the database. However, it is not that cheap to execute 1CSI multi-partition transactions without incurring in extra messages [Vo et al., 2010] that penalize performance. Therefore, it can be assumed that no notion of consistency across data is generally provided for multi-partition transactions, but that data versions are obtained from a valid committed snapshot in each partition. The resulting schedule does not satisfy any of the conditions stated in [Lin et al., 2009]; hence, in the case of multi-partition transactions, we obtain 1CMV schedules.

Apart from this, we have to ensure that transactions executed at the core replicas will eventually get propagated to the rest of replicas inside their associated $\mathcal{RC}$ no matter how many replica failures and network partitions occur, so as to ensure global correctness. Since a replica belonging to a hierarchy layer different than the core layer of an $\mathcal{RC}$ receives its updates from a replica of the upper layer via a FIFO quasi-reliable point-to-point channel, updates are propagated from one hierarchy level to the following, and are received (and therefore executed) in order. This corresponds to the notion of eventual consistency [Fekete and Ramamritham, 2010; Vogels, 2009] (i.e., there is some time point when if update transactions stop then all replicas will converge to the same state). In case the parent (the one that sends updates to the replica) replica fails, this situation will be eventually detected by the metadata manager, which will choose a new replica that will send pending updates to the children replicas. Indeed, this can be understood as a definition of a global liveness property, as the system ensures that in-background update propagation is done correctly.

7.8 EXPERIMENTAL EVALUATION

Although the analytical model and the correctness proof provide an accurate view of the Epidemia's behavior, selecting the proper configuration parameters for such a large-scale system is still challenging. Therefore, this section empirically analyzes how these parameters impact on the Epidemia's maximum throughput using a prototype implementation. Specifically, we aim to analyze the two most relevant parameters that actually make Epidemia unique: the data partitioning scheme (i.e., number of partitions) and the replication protocols (i.e., primary-backup vs. update-everywhere).

7.8.1 *Implementation details*

In order to empirically measure the performance of the proposed system architecture, we have built a prototype using Java 1.6—which provides a sound foundation for future integration of new features (e.g., automatic partitioning [Sancho-Asensio et al., 2014; Porobic et al., 2014])—that covers the basic functionalities of all Epidemia's components (i.e., clients, metadata manager, and replicas).

Clients module. To provide a simple interface to manage invocations to the client library, we have developed an implementation of a JDBC driver that masks calls to the client library [Arrieta-Salinas, 2012]. This JDBC driver provides great flexibility, as it entails the possibility of using the developed prototype as a database in other Java applications by simply changing the JDBC driver that is loaded (using our driver implementation instead of the traditional drivers for other database management systems).

In particular, this JDBC driver has allowed us to run a popular set of benchmarks, named *OLTPBenchmark* [Curino et al., 2012], in order to assess the performance of the developed prototype.

Metadata manager. The implemented version of the metadata manager maintains the metadata repository in main memory and builds the replica hierarchies for the partitions as indicated in the configuration at startup time. This configuration is maintained throughout the execution of each experiment; i.e., the metadata manager does not refresh the system configuration depending on workload variations once the experiment is started, in order to properly evaluate the differences among different scenarios.

Upon receiving a request from a client, the metadata manager examines the partitions that are involved in the requested transaction (which will be those containing any items accessed or modified by the transaction) and selects one replica for each participating partition. A simplified version of the function `chooseReplica` has been implemented. Specifically, this function performs a random selection among the replicas of the replication cluster, taking into account that update transactions must be directed to replicas of the core node that are able to execute update transactions; whereas read-only transactions specify a required freshness level, which is simply a number associated to the maximum hierarchy level that can manage the transaction (hence, the selected replica must belong to a hierarchy level that fulfills the required freshness level). For instance, if the core layer of a replication cluster is managed by a primary-backup replication protocol, all update transactions in which the replication cluster takes part must be directed to the primary replica.

Replicas. Replicas implement the aforementioned primary-backup and update-everywhere replication protocols (see Chapter 2). Each replica holds a PostgreSQL database [PostgreSQL, 2012], that is connected via JDBC.

7.8.2 *Experimental settings*

Our testing configuration consists of eight computers connected in a 100 Mbps switched LAN, where each machine is equipped with an Intel Core 2 Duo processor at 2.13 GHz, 2 GB of RAM and a 250 GB hard disk. All machines run the Linux distribution OpenSuse v11.2 (kernel version 2.6.22.31.8-01), with a Java Virtual Machine 1.6.0 executing the application code. Two additional computers with the same configuration are used for running the clients and the centralized metadata manager instance respectively. Each machine used as a replica holds a local PostgreSQL database (version 8.4.7) [PostgreSQL, 2012], whose configuration options have been tuned so that it behaves as an in-memory only database (i.e., it acts as a cache, without storing the database on disk). Spread 4.0.0 [Stanton, 2005] has been used as GCS, whereas point-to-point communications have been implemented using TCP channels.

The experiments have been run using *OLTPBenchmark* [Curino et al., 2012], a multi-threaded load generator that implements a series of standard OLTP/Web benchmarks and provides several data collection features such as per-transaction-type latency and throughput logs. In particular, we have selected the *OLTPBenchmark* implementation of Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al., 2010], a collection of micro-benchmarks designed to represent data management applications that require high scalability. We have chosen YCSB as the benchmark for the experiments herein presented mainly because its data schema allows a very straightforward partitioning scheme by horizontally splitting the database into subsets of data records. In the YCSB implementation of *OLTPBenchmark*, there exists one table of records with one numeric key and ten text fields. The available set of transactions (where each transaction consists of a single operation) that can be executed against this table are: `read`, which retrieves the record that matches the specified key; `insert`, which inserts a new record; `update`, which updates all the fields of one record with the exception of its key; `delete`, which deletes one record; and `scan`, which reads the set of records whose keys belong to a given interval. The database used for the experiments contains a total of 1 million 1KB records (for a total size of 1GB), distributed among replicas depending on the partitioning scheme.

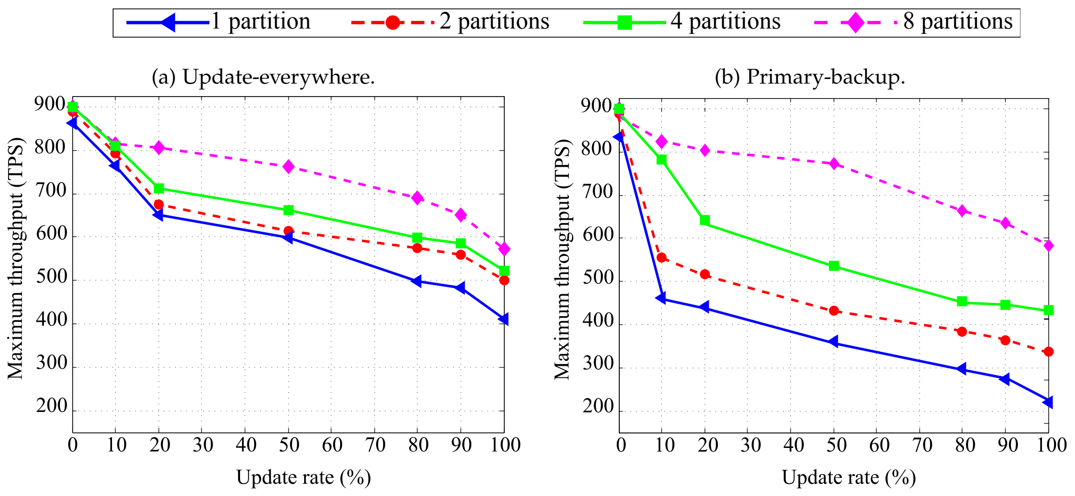


Figure 7.7: Maximum throughput depending on the rate of update vs. read transactions.

7.8.3 *Experiments*

The remainder of this section details the characteristics of the experiments performed using the proposed prototype of Epidemia, as well as the obtained results that validate the feasibility of our approach.

7.8.3.1 *Influence of the partitioning scheme on system's throughput*

In the following experiments, we have assessed the influence of the number of data partitions on system's throughput depending on workload characteristics and the replication protocol used. We have used 100 clients submitting a total workload ranging from 100 to 1000 tps to determine the maximum throughput of the system under different scenarios. We have used 8 replicas to store the database with 4 different data partitioning schemes: i) one partition comprising all data items (the 8 replicas belong to the partition); ii) two partitions, each storing 500K records (4 replicas per partition); iii) four partitions, each storing 250K records (2 replicas per partition) and iv) eight partitions, each storing 125K records (1 replica for each partition). In these experiments, replicas always belong to the core level of their corresponding hierarchy; i.e., all replicas directly take part in the replication protocol, as we are considering only one hierarchy level.

Figure 7.7 shows the maximum throughput obtained for the four different configurations of data partitions mentioned, using (a) update-everywhere and (b) primary-backup as the replication protocols for managing system replicas. In this case, we have used two of the transaction types provided by YCSB: *read* and *update*, which are always single-partition transactions, as each transaction accesses one record. Accessed records are selected according to a Zipfian distribution. The graphics in Figure 7.7 show the influence of the proportion of read/update transactions (ranging from 0% to 100% update transactions) on the system's throughput.

Also,it can be shown that the higher the rate of update transactions, the lower the maximum throughput that can be obtained due to the overhead imposed by update propagation. In addition, both replication protocols have almost the same throughput

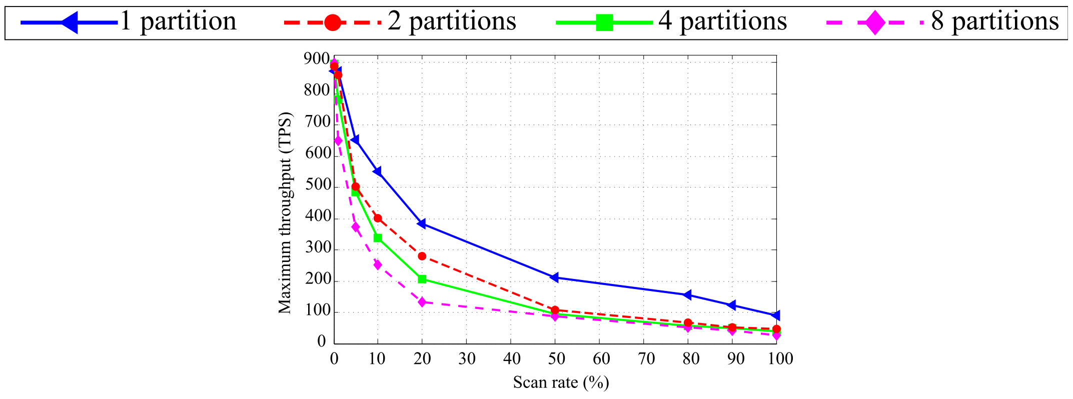


Figure 7.8: Maximum throughput depending on the rate of scan vs. read transactions.

when there is a low rate of updates, as read transactions are handled in the same way. In contrast, primary-backup replication is more costly if there is a high rate of updates, since the primary acts as a bottleneck. We shall remark that as uniform delivery is responsible for the most part of multicast latency, the cost of update multicasts is the same in primary-backup replication, where only FIFO order is needed, and update-everywhere replication, which requires total order. In fact, Spread uses the same level of service for providing uniform reliable multicast, regardless of the ordering guarantees [Stanton, 2005].

As for the influence of the number of partitions on system's throughput, the results of Figure 7.7 verify that, in the case of single-partition transactions, the more data partitions the database is divided into, the more efficient the system is. This is due to the fact that the number of replicas that have to propagate their changes among themselves is inversely proportional to the number of partitions in these experiments, so the cost of propagating changes is lower when we have more partitions in the system. This difference between the throughput of different partitioning schemes is more noticeable in primary-backup replication. In this case, there is one primary replica managing each partition. Therefore, in a configuration with one data partition there is only one primary replica that becomes saturated easily, as it is the only one that can execute update transactions; in contrast, if there is more than one partition the saturation point raises up, as there are more replicas able to handle update transactions for their respective partitions.

The aforementioned results may lead to the wrong conclusion that partitioning the data scheme as much as possible may provide the best performance possible. Actually, we have to bear in mind that the partitioning scheme must also minimize multi-partition transactions, as they are more costly than single-partition transactions. We have verified this statement by repeating the same experiments as before, but in this case using two different types of read-only transactions from the YCSB: *read*, which is always single-partition; and *scan*, which has been slightly modified in order to force it to access several non-consecutive short ranges belonging to different parts of the database within the same transaction.

Figure 7.8 shows the maximum throughput obtained depending on the proportion of read/scan transactions (ranging from 0% to 100% scan transactions), using update-everywhere as the replication protocol (as all transactions are read-only, the replication

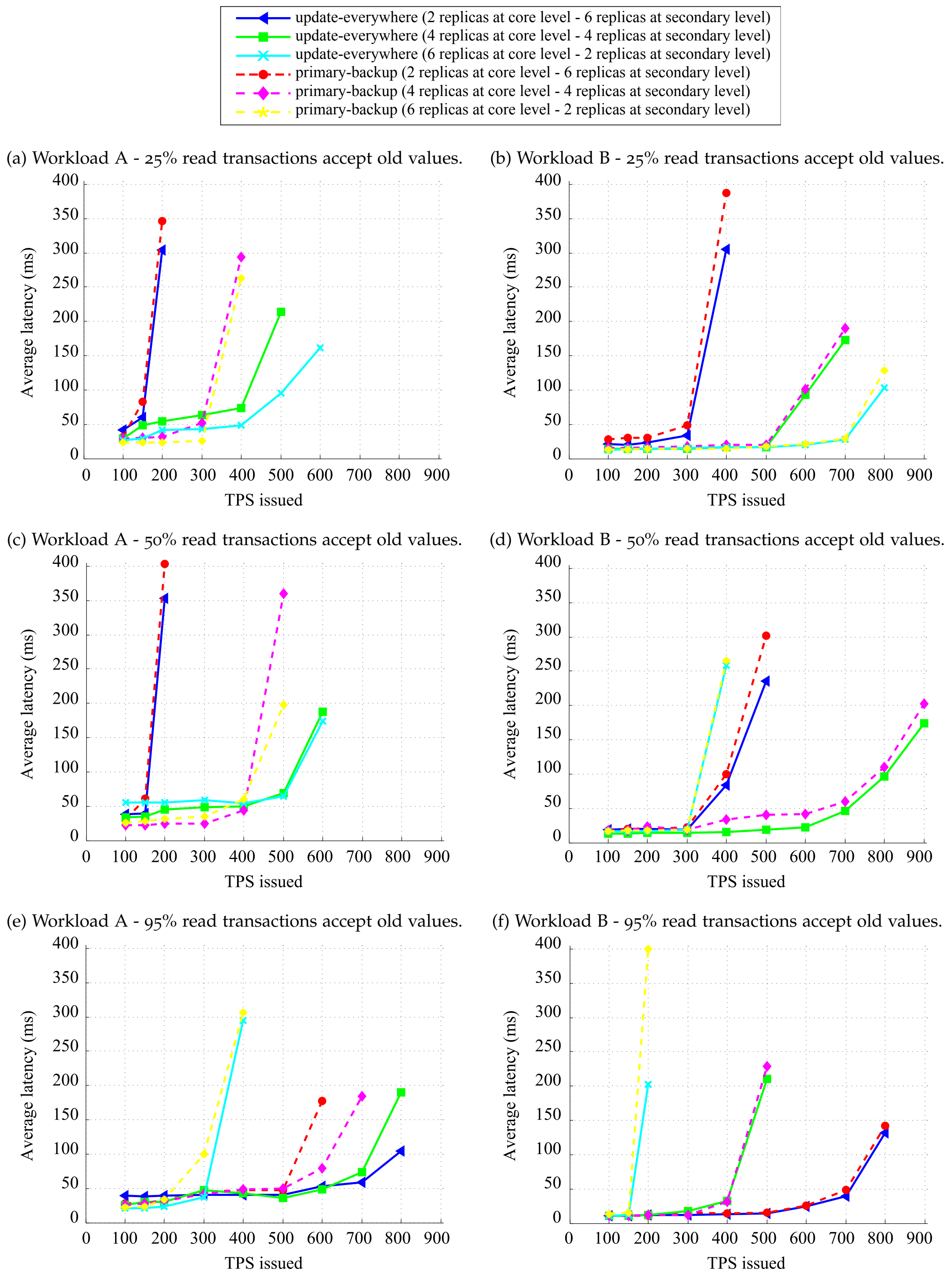


Figure 7.9: Average latency depending on the transactions per second issued for the YCSB workloads A and B with one data partition of 8 replicas.

protocol used has no influence on the results). This figure clearly shows that when there is a high proportion of scan transactions, the throughput is lower in cases where the database is partitioned, because every scan transaction has to access one replica from each partition.

Overall, partitioning the database scheme requires finding the appropriate trade-off to maximize the throughput of single-partition transactions using the available resources while minimizing the need for multi-partition transactions.

7.8.3.2 *Using several hierarchy levels*

In order to test whether the architecture proposed in this chapter can improve the performance of traditional replicated databases by providing additional backup replicas that are asynchronously updated to serve transactions that tolerate a certain degree of staleness, we have configured the system with several hierarchy levels of replicas and have performed a series of experiments using the following YCSB workloads [Cooper et al., 2010]:

- **Workload A.** It models an update-heavy workload (such as recording recent actions in a user session) and is formed by 50% of read transactions and 50% of update transactions that select records according a Zipfian distribution.
- **Workload B.**, It models a read-heavy workload (e.g. photo tagging actions, since adding a tag is an update but most operations are to read tags) and is formed by 95% of read transactions and 5% of update transactions that select records according a Zipfian distribution..

Figure 7.9 shows the average response time (in milliseconds) depending on the number of tps issued to the system for different scenarios using workloads A and B from the YCSB, which have been generated using 100 clients. In this case, we have used one single data partition managed by 8 replicas storing all data items. We have tested different arrangements of the hierarchy: (1) 2 replicas in the core layer and 6 backup replicas in the secondary layer, (2) 4 replicas in the core layer and 4 backup replicas in the secondary layer, and (3) 6 replicas in the core layer and 2 backup replicas in the secondary layer. In these experiments, we have set a predefined freshness level to read transactions, which determines whether the transaction accepts or not stale versions of data items. In particular, we have varied the ratio of read transactions that accept old values, setting this value to 25% for experiments a) and b) of Figure 7.9, 50% for experiments c) and d), and to 95% for experiments e) and f) of the same figure.

In the results of Figure 7.9, the average response time remains quite stable as the number of transactions issued is increased, until a saturation point (that depends on the system settings and the workload characteristics) is reached. At this point, the system is unable to process all incoming requests, so the latency increases dramatically.

In the experiments regarding workload A, the system saturates earlier when using primary-backup replication. This is due to the fact that, although the propagation of updates from the primary to the backup replicas should be cheaper than in the case of update-everywhere replication (because primary-backup does not require total order), Spread actually uses the same mechanism regardless of the ordering guarantees [Stanton, 2005]. As for the experiments for workload B, there are no relevant differences between using update-everywhere or primary-backup as replication protocol for the core layer because 95% of transactions are read-only and both protocols process them identically (by directly executing them).

When most transactions demand a high freshness level (i.e., they must be executed at the replicas of the core layer), the most efficient configurations are those in which most replicas are located in the core layer. In this case, having several hierarchy layers does not entail a relevant improvement on system's performance (see Figure 7.9.a and Figure 7.9.b). Note that the configurations in which there are only two replicas at the core layer are especially inefficient when using workload A with 25% of transactions accepting old values, even using update-everywhere replication, since those two replicas have to execute 50% of incoming transactions (which are update transactions), as well as most read-only transactions and therefore they saturate with a low rate of issued tps.

On the contrary, when most read transactions accept stale values, the more replicas are located in the secondary layer of the hierarchy, the better the system performs. For instance, in Figure 7.9.e and Figure 7.9.f, the configuration with 2 replicas in the core layer using update-everywhere replication and 6 replicas in the secondary layer is the one that outperforms all the others. In the case of Figure 7.9.c and Figure 7.9.d, in which 50% of read transactions tolerate old values, the system performs best when replicas are equally distributed between the core layer and the secondary layer.

Overall, we have confirmed that the existence of several hierarchy levels can contribute to increase system's overall throughput by mitigating the load that replicas participating in the replication protocol are subjected to. Of course, achieving an optimal performance would require an in-depth study of the system's behavior under different scenarios, so as to provide the metadata manager with the needed knowledge base to take the adequate decisions to adapt configuration parameters to the workload at each time.

7.9 SYSTEM DISCUSSION

This chapter has formally presented Epidemia, a distributed storage architecture that combines a cloud-inspired scheme with traditional database replication concepts to provide a highly scalable and available service with transactional support. This section presents a retrospective analysis of Epidema and discusses some of its potential applications.

7.9.1 *Retrospective analysis*

The high scalability and availability featured by Epidemia, is achieved thanks to (1) smartly partitioning data, and (2) using the replication technique based on epidemic updates described in Chapter 5. As a result, Epidemia also provides different consistency levels across a data partitioned scheme, which allows the system adapt to the demands of each client application.

In addition, Epidemia meets the cloud philosophy in the sense that it enables an elastic management of resources (i.e., upgrading and downgrading nodes over the replication chain). This allows the system to dynamically scale out in order to fulfill different client demands, even in the event of load bursts, while optimizing resources usage (i.e, banning or including new nodes on-demand).

Although partitioning may allow some applications to easily unleash all the potential of the elastic cloud, those applications that do not render themselves to an acceptable

Strengths	Weaknesses
- It forwards transactions to the most suitable partition (i.e., load balancing). - It uses epidemic propagation across hierarchically organized partitions. - It selects the most suitable replication strategy for each application. - Resources addition and removal (i.e., elasticity) does not require to synchronize all nodes.	- Multi-partition transactions limit the system throughput. - Applications need to tolerate data partitioning or variable consistency.
Opportunities	**Threats**
- It may push the borders of modern applications that were initially designed to be transactionless. - It opens opportunities for further research on data partitioning and adaptive replication. - Because of the cloud-inspired architecture, Epidemia could be adapted to be a key-value store as existing cloud storage repositories.	- Other cloud-based storage approaches that implement transactional support at the application layer. - Small-sized applications with poorly dynamic behavior. - Restrictive security policies spanning multiple data partitions.

Table 7.1: Strengths, weaknesses, opportunities, and threats of Epidemia

partitioning configuration will still suffer from the well-known scalability limitations of clustered databases, which opens new challenges on many kinds of applications to the cloud.

Moreover, we have shown that the existence of a hierarchy of backups that are asynchronously updated, as done by Epidemia, enables directing transactions that tolerate a certain staleness in the versions of retrieved data items to these backups, which at the same time alleviates the scalability limitations of traditional replicated databases by directing transactions that tolerate a certain staleness in the versions of retrieved data items to these backups.

The many benefits of Epidemia, as well as some drawbacks detected in this study, are summarized in Table 7.1. Specifically, strengths represent the main advantages of Epidemia, weaknesses show its drawbacks, opportunities outline some suggested further lines of investigation, and threats include some optional approaches considered by other methods that could compete with our proposal.

7.9.2 *Potential applications*

The developed implementation used for the experiments provides a sound foundation for future extensions, such as a fully operational metadata manager including decision-making algorithms to dynamically configure the system by splitting or merging data partitions and upgrading or downgrading replicas along hierarchies. Apart from this, replicas could be provided with live migration mechanisms [Das et al., 2011; Elmore et al., 2011] to transfer the necessary information to replicas that join a replication cluster or are upgraded in a hierarchy. Different techniques used in replicated databases, such as recovery using phases [Kemme et al., 2001], could be analized in order to tackle these issues. An extended implementation could also be used for exploring complex formulations for data freshness, such as associating versions with timestamps [Cipar et al., 2012; Lomet et al., 2012] or associating transactions to client sessions to ensure read-your-writes consistency [Vogels, 2009].

From the business point of view, this can be seen as a service where clients pay for consistency; the stronger the consistency obtained, the more expensive the service is. Thus, a Consistency as a Service (CaaS) model could be established, where client applications specify their staleness limit on a per-transaction basis. For instance, this can be applied to web pages in which some parts are seldom updated and do not have strong consistency guarantees (such as the translation of interface messages or some multimedia files) whereas other information (such as account balances or the availability of an item to be purchased) requires strong consistency.

Another practical application of Epidemia is related to the integration of real-time user operations (i.e., OLTP) and tasks involving massive queries that serve as a basis for data mining and decision making tools (i.e., OLAP) into a single storage system. So far, OLTP workloads are typically handled by RDBMS; whereas OLAP tasks are usually executed in data warehouses that periodically collect data from the RDBMS and other sources—which has several inherent limitations such as lack of data freshness in OLAP, redundancy of data storage, high initial capital expenditures, and high maintenance costs [Chen et al., 2010]. On the contrary, our proposal can also serve to support both OLTP and OLAP in an integrated system. On the one hand, OLTP processes can run under different service level agreements according to their needs. For instance, those applications that require strong consistency should access the nodes of the highest level of the replication hierarchy, so as to ensure that any access returns the last updated value. In contrast, there may be other applications that tolerate weaker consistency guarantees; hence, their transactions could involve nodes of lower levels of the hierarchy. On the other hand, OLAP queries should be directed to the lowest levels of the replication hierarchy, in order not to interfere with OLTP workload. The fact that OLAP queries may retrieve data with a certain degree of staleness is usually admissible, given that OLAP is focused on analyzing patterns and trends of massive amounts of information. Moreover, if the OLAP tasks require a certain degree of data freshness, queries could be tagged with a timestamp so that the system would return data with a minimum freshness according to that timestamp.

Contribution.

1. Enumeration of the challenges related to provide transactional support on the cloud.

2. Presentation of Epidemia—a distributed storage architecture featuring a hybrid approach that combines classic database replication techniques with a cloud-inspired

infrastructure to provide transactional support as well as high availability—and its design rationale.

3. Formal specification of Epidemia's system components and validation of its formal correctness.
4. Development of an analytical model to compute the scale out factor of partitioned and transactional databases.
5. Experimental evaluation on the effects that the replication protocols and partitioning schemes have on Epidemia's throughput.

8 SUMMARY, CONCLUSIONS, AND FURTHER WORK

Summary. This thesis has addressed several research goals involving distributed databases, cloud storage repositories, and transactional support. This chapter wraps up the main findings of the dissertation, elaborates the obtained conclusions, and discusses some future research directions.

"Imagination is more important than knowledge. For knowledge is limited to all we now know and understand, while imagination embraces the entire world, and all there ever will be to know and understand"
— Albert Einstein, 1931.

8.1 INTRODUCTION

This thesis has investigated transactional support as an effective way to improve the features provided by current cloud storage repositories. Specifically, we have addressed two important challenges not only for cloud storage but also for classic databases: reasoning scalability limitations of distributed databases and allowing transaction-based applications to benefit from the cloud philosophy. To address the first challenge, we started building a simulator to analyze the behavior of replication techniques and concurrency control algorithms in a transactional database, which allowed us to see the source of their stringent scalability and suggested some strategies to improve it. To approach the second challenge, we put together the lessons previously learnt from static transactional environments, and proposed a custom key-value data store—inspired by existing Big Data and cloud approaches—to face a real-world problem concerning massive data storage in Smart Grids. The development of this proposal exhibited the limitations of existing cloud technologies regarding transactional support, which drove us to further evolve it into what we coined as Epidema; a cloud-inspired transactional database that dynamically trades consistency for availability and scalability.

The purpose of this chapter is to summarize the work carried on the consecution of each research goal, highlight the obtained conclusions, and provide some future work directions that could be considered in light of the outcomes provided by this dissertation.

8.2 SUMMARY AND CONCLUSIONS

Over the last few years, researchers have been concerned on limiting the features provided by traditional databases (e.g., relational algebra, data normalization, ACID properties) to address the scalability challenges posed by Big Data. Indeed, the ever-growing amount of information to be stored and efficiently retrieved by modern applications is shifting aside the appealing features provided by classic databases. In fact, we are paradoxically experiencing a regression process in which practitioners are removing databases functionalities—rather than including new ones—in order to push the scalability frontiers to the (virtual) infinite. However, there are still several applications that, despite their intrinsic need on dealing with massive amounts of data, cannot resign from their transactional nature and, thus, are unable to benefit from the latest advances in this field. In such a context, we started this thesis by identifying two critical challenges in the field of distributed databases:

1. Scalability in transactional distributed databases.
2. Transactional support on cloud-based storage repositories.

These two challenges were motivated by the structural design of classic transactional databases. In fact, these systems—created on the early eighties—were not initially designed to support huge amounts of data. Instead, they favored handling a large number of users by replicating data in several machines and, thus, obtaining high availability, which was pretty reasonable by that time because it was more likely that the number of users grew faster than the amount of data. Additionally, the overhead associated to information handling and indexing was successfully supported by smart data structures and computationally powerful RDBMSs. This situation led database practitioners on enhancing and improving the main congestion sources of those systems: concurrency control and replication protocols.

However, when data volumes (and number of users) rocketed, these classic systems were unable to scale accordingly due to the fact that data was tightly coupled by relational schemes and transactions. Therefore, recent highly scalable cloud-based systems have taken a shotgun approach and suggest to store simple data structures (i.e., key-value pairs)—that do not actually require transactional support—with limited concurrency control and replication facilities in order to achieve (virtually) infinite scalability.

Alternatively, we aimed to combine the benefits of both solutions (i.e., transactional support and cloud-based storage) to address the proposed challenges, which drove us to articulate the following four main goals:

1. Revise concurrency control protocols and replication techniques to compare their performance.
2. Analyze cloud-based storage repositories.
3. Apply the cloud storage paradigm to solve current scalability issues in industry.
4. Design and implement a storage infrastructure able to provide transactional support on the cloud.

A summary of the work conducted under each objective and the conclusions extracted from each point are provided in what follows:

Revise concurrency control protocols and replication techniques to compare their performance. The first objective of this dissertation proposed to examine existing concurrency control and replication protocols to find out the source of their limited scalability. Therefore, we initially reviewed the theory associated to distributed databases and protocols. Upon this theory, we created a custom simulator for database protocols and benchmarked three well-known concurrency control and replication techniques: NDCC, BRP, BOCC. Specifically, we evaluated (1) the effects of adding more nodes to replicate data, (2) the maximum number of tps that each protocol could afford on a given load, (3) how partial replication affected the system performance, and (4) the impact of the transactions size on the system throughput.

This empirical analysis provided several insights into the consequences of selecting the proper replication strategy, concurrency control protocol, data partitioning scheme, and transactions size for a static distributed database. First, we observed that the number of messages exchanged by replication and concurrency control protocols to ensure ACID properties alarmingly grows with the number of servers in which a data object is replicated. Therefore, it is fundamental to exploit data locality whenever possible in order to avoid flooding the whole network. Second, we saw that communication networks (specially those ones with low capacity that link geographically distant locations) may act as a bottleneck for those protocols that demand a high number of network messages. Thus, it is important to minimize the size and number of this messages. Third, as already suggested by Kemme [2000]; Serrano et al. [2007]; Sutra and Shapiro [2008], we corroborated that partitioning data can greatly increase the database performance as long as multi-partition transactions are avoided. Finally, we found that reducing the size of transactions greatly increases the database throughput. Indeed, the less data items involved in a transaction, the lower chance of conflicts (i.e., probability of two transactions accessing the same object at the same time) will be. Overall, we concluded that transactions together with replication and concurrency control protocols used to ensure ACID properties were the main cause of the stringent scalability issues of classic databases.

These findings helped us to understand the rationale behind cloud storage repositories since they exploit transactions size (i.e., key-value pairs and NoSQL), data partitioning, and non-ACID semantic properties (i.e., BASE) to provide highly scalable multi-tenant systems.

Analyze cloud-based storage repositories. After identifying the main causes that prevent classic databases from handling huge amounts of data, we explored cloud storage systems that promise to be an effective alternative for poorly scalable RDBMSs. Therefore, we initially analyzed the requirements stated by the cloud philosophy and decomposed the most significant state-of-the-art cloud-based data repositories, which allowed us to draw an abstract reference model for highly scalable storage systems. Additionally, we analyzed the techniques used by each system to address a broad range of highly-scalable applications.

In this process, we observed that cloud systems are able to elastically scale on demand thanks to their reasonably simple architectural design: a small set of metadata nodes that manage an unlimited number of storage servers and forward user queries to the appropriate storage facility. Therefore, the metadata manager can be best seen as a central entity in charge of balancing the load among storage servers, which at the same time (1) greatly reduces the synchronization overhead associated to the addition or removal of physical storage facilities, (2) simplifies the concurrency control process, and (3) allows to dynamically move and replicate data wherever it is needed. To alleviate the load of the metadata manager, cloud storage repositories typically support simple data structures (i.e., key-value pairs) with no transactional support, which minimizes the size, complexity, and computation time of user requests. Additionally, to further exploit this potentially bottleneck prone architecture—specially when deployed in wide span areas—and as a response to the constraints stated by Brewer's theorem, these systems offer a weak form of the ACID properties referred to as BASE, which basically consists in prioritizing availability in front of durability (i.e., storing data in main memory) and consistency (i.e., using weak consistency models such as eventual consistency). This situation has driven practitioners to develop the logic associated to the properties that have been neglected by cloud repositories into the application layer, which complicates the applications design and provides suboptimal—but, so far, functional—results. Overall, we concluded that highly scalable and available storage systems are feasible as long as the properties of transactional databases can be relaxed.

This analysis corroborated the conclusions extracted from the previous goal regarding the key factors that limited transactional databases scalability. Also, it provided us with the fundamentals to combine both techniques—classic databases and cloud storage—and address massive data storage in real-world problems.

Apply the cloud storage paradigm to solve current scalability issues in industry. To empirically evaluate the advantages claimed by cloud repositories and observe the practical implications of their design, we exposed them to a real-world scenario. First, we conducted a prototype experiment with the SABI dataset aimed to assess the deployment, development, and maintenance cost of the most representative cloud storage platform: Hadoop [White, 2011]. Afterwards, we decided to address large-scale data storage in Smart Grids, which represents a latent problem in industry. In this regard, we proposed a storage architecture, inspired by existing cloud approaches, that incorporated replication strategies used in classic databases.

The prototype experiment conducted with Hadoop allowed us to experience the tedious burden associated to this young and continuously evolving technology, which prevents itself from being deployed in industrial scenarios. Also, in this experiment we explored the MapReduce distributed computation paradigm, which allowed us to see the importance of minimizing the number of network messages and exploiting data locality in Big Data scenarios. When exporting the lessons learnt with Hadoop to Smart Grids, we found that there is a big gap between research contributions and industry demands. Therefore, we proposed an alternative distributed data repository specifically designed to meet the Smart Grid requirements. We observed that high scalability and availability in dynamic environments can be achieved by (1) inheriting the simple system architecture from cloud repositories, (2) keeping reasonably small data partitions, and (3) epidemically propagating updates among these data islands using classic replication strategies. This hybrid strategy allows the system to offer elasticity at a low cost (i.e., resources addition or removal only affects to a data partition) and provide different consistency degrees. Also, by dynamically adjusting the configuration of data partitions, this strategy is able to naturally offer *k*-safety [Stonebraker and Weisberg, 2013] at the price of altering data availability. Overall, we concluded that (1) existing cloud storage repositories are often too generic for being applied to real situations, and (2) features from classic databases (e.g., replication protocols, ACID properties) can be combined with the cloud philosophy.

Combining cloud storage and classic databases, as we did for Smart Grids, showed us that there is still space for revisiting old static distributed database techniques and adapt them to the modern challenges in data storage.

Design and implement a storage infrastructure able to provide transactional support on the cloud. Transactional support is a key enabler to further extend the number of applications that can take advantage of the cloud storage paradigm. Therefore, we collected the main lessons extracted when designing the custom key-value repository for Smart Grids and conceived Epidemia, a general purpose storage solution able to provide transactional support on the cloud. Epidemia keeps combining the advantages of cloud storage (i.e., high scalability and availability) with the features of classic distributed databases (i.e., transactions, ACID properties, replication protocols, concurrency control) allowing to dynamically adapt its configuration to applications demands. Indeed, providing transactional support on the cloud is not a novel idea by itself, but the way in which Epidemia has done it makes itself a unique approach.

Upon implementing Epidemia, we observed that epidemically propagating update operations along a replication chain results on an intrinsically hierarchical organization of data versions. This allows to conduct OLAP and OLTP operations simultaneously at different levels of the hierarchy, which greatly reduces the overhead associated to Extract, Transform, and Load (ETL) processes (i.e., storing data twice, moving information to a data warehouse) typically carried by Business Intelligence applications. Additionally, we saw that dynamically adapting the replication protocol and partitioning scheme—despite the cost of synchronizing all replicas [Elmore et al., 2011; Das et al., 2011]—according to the workload characteristics allows the system to cope with realistic scenarios and, thus, better fit to the cloud philosophy (i.e., provide elastic resources on demand). However, there is still a long way to do on envisaging the proper moment to carry such reconfigurations [Sancho-Asensio et al., 2014] and preventing the system from being continuously reconfigured, which would limit its throughput. Also, developing an integral solution from its conception to the implementation of a prototype, made us understand the reasoning behind most existing approaches that use third-party cloud storage solutions to provide transactional support (e.g., ElaSTraS [Das et al., 2010b], Omid [Gómez Ferro et al., 2014]), which allows them to speed up the time to market of their solutions. Overall, Epidemia is a practical solution that puts together all the knowledge gained upon the consecution of the previous goals in this dissertation.

To sum up, the contributions of this dissertation emphasize that the techniques used by classic transactional databases—and their associated features—should not be put aside when facing modern challenges in data storage. The insights provided along this work show that a hybrid approach between cloud storage and transactional databases can improve the individual limitations of each domain (i.e., scalability and potential data-driven applications). Furthermore, we have also illustrated the compatibility between cloud storage and transactional schemes, which allows to obtain great results if balanced properly. For this reason, classic databases probably have much to say in the future of data storage and cloud computing.

The lessons learned longwise the realization of this thesis have also served to define new objectives that will be hopefully approached in future works. The next section defines and discusses them.

8.3 FUTURE WORK

This work embraces several topics in the field of distributed systems and data storage. For each topic, we have identified some open issues that demand further research that could improve the ideas herein presented. These future work directions are presented thereafter following the chapter structure of the dissertation.

Chapter 2. So far, distributed storage systems have been typically designed to be unaware from the features provided by the network communications layer. For instance, it is very common to find concurrency control protocols that implement complex message exchanging mechanisms to ensure a FIFO order on messages delivery [Chockler et al., 2001]. However, this could be easily guaranteed by existing OSI layer 4 protocols. Another sound example of this situation can be found on the fault detection protocols; these protocols generally result in several network messages that poll the status of each node and use timeout policies to infer the status of the distributed system, which degrades the network performance and becomes critical when facing large-scale distributed storage systems. However, routing protocols are specifically designed to (1) detect and isolate faulty nodes, (2) find the best communication path between two sites, and (3) afford dynamic environments. Indeed, low OSI levels own updated and precise information about the communication network status that could greatly assist on the duties of the application layer. This situation suggests an integrated concurrency control and replication protocol that uses the information from low OSI layers to find the optimal strategy to replicate data. Such symbiosis would alleviate the overhead associated to data layer tasks when facing dynamic environments and, thus, improve the overall system performance.

Chapter 3. The proposed protocol simulator for distributed databases allows a rapid prototyping and evaluation of replication protocols and concurrency control algorithms. This has allowed us to see that the selected policy used to decide which transaction(s) must abort in case of conflict, has a great impact on the system throughput. Therefore, it might be worth to further analyze the effects of first committed wins versus first updated wins concurrency control strategies to complement the extensive research conducted on replication protocols [Wiesmann and Schiper, 2005]. Specifically, this research should come up with a relation between the data access pattern and the best concurrency control protocol to be used.

Chapter 4. MapReduce—and its modern version YARN—is a powerful tool to conduct distributed computations with massive amounts of data by exploiting data locality and the intrinsic features of the underlying cloud-based storage filesystem. However, coming up with the appropriate configuration parameters—from both the filesystem and MapReduce—to minimize the computation time is, at the moment, a matter of trial and error tests. Therefore, we propose to further analyze (1) the impact of each configuration parameter on the final system performance, and (2) the different map and reduce types of task. With this analysis, it might be possible to build a model cost and, thanks to machine learning tools, build a ruleset to guide practitioners on properly configuring MapReduce environments.

Chapter 5. Further exploring the possibilities that the proposed distributed storage architecture for the Smart Grid has on conducting data computation duties might be definitely worthy. Alternatively, it might be worth to consider another approach to both store and compute data: graph databases. In fact, graph databases might interpret each IED/I-Dev as a node from a graph with structured data whose schema might be different on a per device basis. This approach could (1) speed-up the data computation and search time (i.e., graph databases are designed to be efficient at traversing the node space), (2) adapt to the evolving nature of the smart functions (recall that Smart Grids are still on early stages of their conception), and (3) reduce the overhead associated to denormalizing data. Nonetheless, taking into account the vast amount of data generated by Smart Grids, the difficulties associated to graph sharding should be carefully addressed.

Chapter 6. Smart Grids integrate several devices from different vendors that run different protocols and policies in order to reach a common goal: bring together energy delivery and smart services. Providing a global monitoring and management interface for such a large-scale infrastructure will become essential as soon as it becomes a reality beyond current feasibility tests [Repo et al., 2011]. A similar problem has been already faced by the Internet of Things in which many heterogeneous technologies interact under a common environment. Therefore, we propose to apply the latest advances from the Internet of Things to provide a unified and ubiquitous management interface for Smart Grids.

Chapter 7. The future work directions derived from this chapter are twofold. One the one hand, we acknowledge the need of conducting an extensive stress test of Epidemia in a large-scale scenario with hundreds of machines (e.g., Amazon EC2), to (1) further asses up to what extent transactional support penalizes on scalability, (2) analyze the performance effects when establishing a data security policy, and (3) compare the obtained analytical results with real-world scenarios.

On the other hand, we suggest to have a closer look at one of the key configuration parameters in Epidemia: data partitioning. Designing a partitioning scheme requires a deep knowledge of data and their access patterns, which is not feasible to obtain in several situations due to the dynamic and complex conditions of the environment [Curino et al., 2010]. Therefore, it might be worth to use machine learning techniques to discover regular-occurring patterns beneath the data without doing any a priori assumptions concerning their underlying structure (i.e., unsupervised learning). Moreover, this intelligent system—integrated at the metadata manager module of Epidemia—should (1) automatically learn the side effects—in terms of performance gain/loss, service downtime, and migration cost—associated to reconfiguring the partitioning scheme, and (2) come up with the best moment to reconfigure data partitions.

BIBLIOGRAPHY

[Abdelguerfi and Wong, 1998] Mahdi Abdelguerfi and Kam-Fai Wong. *Parallel database techniques*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1998. ISBN 0818683988.

[Acampora et al., 2011] Giovanni Acampora, Vincenzo Loia, and Autilia Vitiello. Exploiting timed automata based fuzzy controllers for voltage regulation in Smart Grids. In *Proceedings of the IEEE International Conference on Fuzzy Systems*, pages 223–230, 2011. ISBN 978-1-4244-7315-1.

[Adya, 1999] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology, 1999.

[Adya et al., 2000] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, pages 67–78, 2000.

[AEP Ohio, 2011] AEP Ohio. About the gridSMART project. In *A new way to think about electricity*. Web. Retrieved February 2011. `www.aepohio.com/save/demoproject/about/Default.aspx`, 2011.

[Agrawal et al., 1994] Divyakant Agrawal, Amr El Abbadi, and AE Lang. The performance of protocols based on locks with ordered sharing. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):805–818, 1994.

[Agrawal et al., 2011] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: Current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533, 2011. ISBN 978-1-4503-0528-0.

[Aguilera et al., 2009] Marcos Kawazoe Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3), 2009.

[Al-Jumah et al., 2000] NB Al-Jumah, Hossam Hassanein, and Mohamed El-Sharkawi. Implementation and modeling of two-phase locking concurrency controla performance study. *Information & Software Technology*, 42(4):257–273, 2000.

[Albino, 2008] Pablo Murta Baião Albino. Eficiencia y productividad de las cooperativas de crédito españolas frente al desafío de la desintermediación financiera. In *INTERNATIONAL, C. E. A. C. (Ed.) Innovation and management: Answers to the great challenges of public, social economy and cooperative enterprises*, 2008.

[Alonso et al., 1996] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, Mohan Kamath, Roger Günthör, and Chandrasekaran Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the 12th International Conference on Data Engineering*, pages 574–581, 1996. ISBN 0-8186-7240-4.

[Alsberg and Day, 1976] Peter Alsberg and John Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, 1976.

[Amir and Tutu, 2002] Yair Amir and Ciprian Tutu. From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 494–503. IEEE, 2002.

[Amir et al., 1997] Yair Amir, Gregory Chockler, Danny Dolev, and Roman Vitenberg. Efficient state transfer in partitionable environments. Technical report, Institute of Computer Science, The Hebrew University, 1997.

[Armbrust et al., 2009] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Howard Katz, Andrew Konwinski, Gunho Lee, David Andrew Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[Armbrust et al., 2010] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4): 50–58, April 2010. ISSN 0001-0782.

[Armendáriz-Iñigo, 2006] José Enrique Armendáriz-Iñigo. *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Departamento de Matemática e Informática–Universidad Pública de Navarra, 2006.

[Armendáriz-Iñigo et al., 2007] José Enrique Armendáriz-Iñigo, José Ramón Juárez-Rodríguez, José Ramón González de Mendívil, Hendrik Decker, and Francesc Daniel Muñoz-Escoí. *k*-bound GSI: A flexible database replication protocol. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 556–560, 2007. ISBN 1-59593-480-4.

[Armendáriz-Iñigo et al., 2008] José Enrique Armendáriz-Iñigo, Augusto Mauch-Goya, José Ramón González de Mendívil, and Francesc Daniel Muñoz-Escoí. SIPRe: A partial database replication protocol with SI replicas. In *Proceedings of the 2008 ACM Symposium on Applied Computing, SAC 2008*, pages 2181–2185, 2008. ISBN 978-1-59593-753-7.

[Arnold, 2011] George Arnold. Challenges and opportunities in Smart Grid: A position article. *Proceedings of the IEEE*, 99(6):922–926, June 2011.

[Arrieta-Salinas, 2012] Itziar Arrieta-Salinas. *Study and development of a transactional database system on a cloud environment*. Master's thesis, Universidad Pública de Navarra, Spain, 2012.

[Babot, 2009] Francesc Xavier Babot. *Contributions to formal communication elimination for system models with explicit parallelism*. PhD thesis, Universitat Ramon Llull. Enginyeria i Arquitectura La Salle, 2009.

[Baker et al., 2011] Jason Baker, Chris Bond, James Corbett, Jeffrey John Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim

Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[Bartoli, 1999] Alberto Bartoli. Reliable distributed programming in asynchronous distributed systems with group communication. Technical report, Università di Trieste, Trieste, Italy, 1999.

[Bartoli, 2004] Alberto Bartoli. Implementing a replicated service with group communication. *Journal of Systems Architecture*, 50(8):493–519, 2004.

[Beltran, 2010] Miquel Beltran. *Modelat i verificació de sistemes distribuïts*. Universitat Ramon Llull. Enginyeria i Arquitectura La Salle, 2010. ISBN 978-84-937712-9-4.

[Berenson et al., 2007] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *Computing Research Repository*, abs/cs/0701157, 2007.

[Bernabé-Gisbert et al., 2008] Josep Maria Bernabé-Gisbert, Vaide Zuikeviciute, Francesc Daniel Muñoz-Escoí, and Fernando Pedone. A probabilistic analysis of snapshot isolation with partial replication. In *Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems*, pages 249–258, 2008.

[Bernstein and Das, 2013] Philip Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 923–928. ACM, 2013. ISBN 978-1-4503-2037-5.

[Bernstein and Goodman, 1981] Philip Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[Bernstein and Goodman, 1983] Philip Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the 2nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 114–122, 1983. ISBN 0-89791-110-5.

[Bernstein et al., 1987] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[Birman, 2012] Kenneth Birman. *Reliable distributed systems*. Manning Publications Co., 2012.

[Birman and Joseph, 1987] Kenneth Birman and Thomas Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[Botan et al., 2010] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée Miller, and Nesime Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the Very Large Data Bases Endowment*, 3(1): 232–243, 2010.

[Bouhafs et al., 2012] Faycal Bouhafs, Michael Mackay, and Madjid Merabti. Links to the future. *IEEE Power & Energy*, 10(1):24–32, 2012.

[Brantner et al., 2008] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 251–264, 2008.

[Brewer, 2000] Eric Allen Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, page 7, 2000. ISBN 1-58113-183-6.

[Brewer, 2012] Eric Allen Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45:23–29, 2012. ISSN 0018-9162.

[Brown, 2008] Richard Brown. Impact of Smart Grid on distribution system design. *Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, pages 1–4, 2008.

[Brownbridge et al., 1982] David Robert Brownbridge, Lindsay Forsyth Marshall, and Brian Randell. The newcastle connection or unixes of the world unite! *Software Practice and Experience*, 12(12):1147–1162, 1982.

[Bull et al., 2004] Larry Bull, Andy Tomlinson, John Addison, and Benjamin Heydecker. Towards distributed adaptive control for road traffic junction signals using learning classifier systems. In *Applications of Learning Classifier Systems*, pages 276–299, 2004.

[Bull et al., 2007] Larry Bull, Matthew Studley, Anthony Bagnall, and Ian Whittley. Learning classifier system ensembles with rule-sharing. *IEEE Transactions on Evolutionary Computation*, 11(4):496–502, 2007.

[Burckhardt et al., 2012] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Proceedings of the 21st European Symposium on Programming*, pages 67–86, 2012. ISBN 978-3-642-28868-5.

[Bureau van Dijk, 2010] Bureau van Dijk. Sabi. In *SABI database*. Web. Retrieved February 2011. `http://sabi.bvdep.com`, 2010.

[Butz, 2006] Martin Butz. *Rule-based evolutionary online learning systems – A principled approach to LCS analysis and design*, volume 191 of *Studies in Fuzziness and Soft Computing*. Springer, 2006. ISBN 978-3-540-25379-2.

[Campbell et al., 2010] David Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1021–1024, 2010.

[Campos et al., 2014] Filipe Campos, Miguel Matos, José Orlando Pereira, and David Rua. A peer-to-peer service architecture for the Smart Grid. In *Proceedings of the IEEE 14th International Conference on Peer-to-Peer Computing*, London, United Kingdom, September 2014.

[Cao et al., 2011] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. ES^2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of the 27th International Conference on Data Engineering*, pages 291–302, 2011. ISBN 978-1-4244-8958-9.

[Carcano et al., 2011] Andrea Carcano, Alessio Coletta, Michele Guglielmi, Marcelo Masera, Igor Nai Fovino, and Alberto Trombetta. A multidimensional critical state analysis for detecting intrusions in scada systems. *IEEE Transactions on Industrial Informatics*, 7(2):179–186, 2011.

[Carey and Livny, 1991] Michael Carey and Miron Livny. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database Systems*, 16(4):703–746, December 1991. ISSN 0362-5915.

[Carse et al., 1996] Brian Carse, Terence Claus Fogarty, and Alistair Munro. Distributed adaptive routing control in communication networks using a temporal fuzzy classifier system. In *Proceedings of thee IEEE International Conference on Fuzzy Systems*, volume 3, pages 2201–2207, 1996.

[Carstoiu et al., 2010] Dorin Carstoiu, Elena Lepadatu, and Mihai Gaspar. Performances evaluation Hbase – non SQL Database. *International Journal of Advances in Computing Technology*, 2(5):42–52, 2010.

[Casteigts, 2010] Arnaud Casteigts. The JBotSim library. *Computing Research Repository*, abs/1001.1435, 2010.

[Castro and Liskov, 2002] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[Cattell, 2010] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39 (4):12–27, 2010.

[Chang et al., 2008] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[Chen et al., 2010] Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Providing scalable database services on the cloud. In *Proceedings of the 11th International Conference on Web Information System Engineering*, pages 1–19, 2010. ISBN 978-3-642-17615-9.

[Cheung et al., 2012] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew Myers. Automatic partitioning of database applications. *Proceedings of the Very Large Data Bases Endowment*, 5(11):1471–1482, 2012.

[Chockler et al., 2001] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33 (4):427–469, 2001.

[Chuang and McGranaghan, 2008] Angela Chuang and Mark McGranaghan. Functions of a local controller to coordinate distributed resources in a smart grid. *Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, pages 1–6, 2008. ISSN 1932-5517.

[Cipar et al., 2012] James Cipar, Gregory Ganger, Kimberly Keeton, Charles Morrey III, Craig Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the Seventh European Conference on Computer Systems*, pages 169–182, 2012. ISBN 978-1-4503-1223-3.

[Cooper et al., 2008] Brian Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the Very Large Data Bases Endowment*, 1(2):1277–1288, 2008.

[Cooper et al., 2010] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[Cooper et al., 2009] Brian Frank Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James Jay Kistler, PPS Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, and Raymie Stata. Building a cloud for Yahoo! *IEEE Data Engineering Bulletin*, 32(1):36–43, 2009.

[Corbett et al., 2012] James Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

[Cristian, 1991] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[Curino et al., 2010] Carlo Curino, Yang Zhang, Evan Jones, and Samuel Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the Very Large Data Bases Endowment*, 3(1):48–57, 2010.

[Curino et al., 2011a] Carlo Curino, Evan Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 313–324, 2011a.

[Curino et al., 2011b] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database service for the cloud. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 235–240, 2011b.

[Curino et al., 2012] Carlo Curino, Djellel Eddine Difallah, Andrew Pavlo, and Philippe Cudré-Mauroux. Benchmarking OLTP/Web databases in the cloud: The OLTP-bench framework. In *Proceedings of the 4th International Workshop on Cloud Data Management*, pages 17–20, 2012. ISBN 978-1-4503-1708-5.

[Das, 2011] Sudipto Das. *Scalable and elastic transactional data stores for cloud computing platforms*. PhD thesis, University of California, Santa Barbara, December 2011.

[Das et al., 2010a] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 163–174, 2010a. ISBN 978-1-4503-0036-0.

[Das et al., 2010b] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic transactional data store in the cloud. *Computing Research Repository*, abs/1008.3751, 2010b.

[Das et al., 2011] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the Very Large Data Bases Endowment*, 4(8):494–505, 2011.

[Daudjee and Salem, 2006] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 715–726, 2006. ISBN 1-59593-385-9.

[de Sousa et al., 2001] António Luís Pinto Ferreira de Sousa, Rui Carlos Oliveira, Francisco Moura, and Fernando Pedone. Partial replication in the database state machine. In *Proceedings of the 2001 IEEE International Symposium on Network Computing and Applications*, pages 298–309, 2001. ISBN 0-7695-1432-4.

[Dean and Ghemawat, 2010] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[DeCandia et al., 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 205–220, 2007. ISBN 978-1-59593-591-5.

[Défago and Schiper, 2004] Xavier Défago and André Schiper. Semi-passive replication and lazy consensus. *Journal of Parallel and Distributed Computing*, 64(12):1380–1398, 2004.

[Défago et al., 2004] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[Della Giustina et al., 2011] Davide Della Giustina, Sami Repo, Stefano Zanini, and Lucio Cremaschini. ICT architecture for an integrated distribution network monitoring. In *Applied Measurements for Power Systems (AMPS), 2011 IEEE International Workshop on*, pages 102 –106, sept. 2011.

[Dwork et al., 1988] Cynthia Dwork, Nancy Ann Lynch, and Larry Joseph Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[Elmore et al., 2011] Aaron Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 301–312, 2011.

[Elnikety et al., 2005] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84, 2005. ISBN 0-7695-2463-X.

[Eramo et al., 2008] Vincenzo Eramo, Michele Listanti, and Antonio Cianfrani. Multi-path ospf performance of a software router in a link failure scenario. In *Proceedings of the 4th International Telecommunication Networking Workshop on QoS in Multiservice IP Networks*, 2008.

[Fekete and Ramamritham, 2010] Alan David Fekete and Krithi Ramamritham. Consistency models for replicated data. In *Replication*, pages 1–17, 2010.

[Florescu and Kossmann, 2009] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[Fox et al., 1997] Armando Fox, Steven Gribble, Yatin Chawathe, Eric Allen Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91, 1997.

[Franaszek et al., 1992] Peter Anthony Franaszek, John Robinson, and Alexander Thomasian. Concurrency control for high contention environments. *ACM Transactions on Database Systems*, 17(2):304–345, 1992.

[Friedman and van Renesse, 1996] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in horus. In *Proceedings of the 15th IEEE International Symposium on Reliable Distributed Systems*, pages 140–149, 1996.

[Galli et al., 2010] Stefano Galli, Anna Scaglione, and Zhifang Wang. For the grid and through the grid: The role of power line communications in the Smart Grid. *Computer Research Repository*, abs/1010.1973, 2010.

[Ghemawat et al., 2003] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003. ISBN 1-58113-757-5.

[Gifford, 1979] David Gifford. Weighted voting for replicated data. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, pages 150–162, 1979.

[Gilbert and Lynch, 2002] Seth Gilbert and Nancy Ann Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[Golab and Johnson, 2011] Lukasz Golab and Theodore Johnson. Consistency in a stream warehouse. In *In Online Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 114–122, 2011.

[Gómez Ferro et al., 2014] Daniel Gómez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *Proceedings of the IEEE 30th International Conference on Data Engineering*, pages 676–687, 2014. ISBN 978-1-4799-3480-5.

[Gray, 1980] Jim Gray. A transaction model. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 282–298, 1980. ISBN 3-540-10003-2.

[Gray et al., 1996] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In Hosagrahar Visvesvaraya Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM Press, 1996.

[Guerraoui and Schiper, 1994] Rachid Guerraoui and André Schiper. Transaction model vs. virtual synchrony model: Bridging the gap. In *Proceedings of the International Workshop on Theory and Practice in Distributed Systems*, pages 121–132, 1994. ISBN 3-540-60042-6.

[Gungor et al., 2013] Vehbi Cagri Gungor, Dilan Sahin, Taskin Kocak, Salih Ergüt, Concettina Buccella, Carlo Cecati, and Gerhard Hancke. A survey on smart grid potential applications and communication requirements. *IEEE Transactions on Industrial Informatics*, 9(1):28–42, 2013.

[Guzmán et al., 2009] Isidoro Guzmán, Narciso Arcas, Rino Ghelfi, and Sergio Rivaroli. Technical efficiency in the fresh fruit and vegetable sector: A comparison study of Italian and Spanish firms. *Fruits*, 64(4):243–252, July-August 2009.

[Hadzilacos and Toueg, 1994] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca, NY, USA, 1994.

[Härder and Reuter, 1983] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[Harter et al., 2014] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea Carol Arpaci-Dusseau, and Remzi Hussein Arpaci-Dusseau. Analysis of hdfs under hbase: A Facebook messages case study. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014*, pages 199–212, 2014. ISBN 978-1-931971-08-9.

[Hernández-Cánovas and Martínez-Solano, 2010] Ginés Hernández-Cánovas and Pedro Martínez-Solano. Relationship lending and SME financing in the continental European bank-based system. *Small Business Economics*, 34(4):465–482, May 2010.

[Holland, 1992] John Holland. *Adaptation in Natural and Artificial Systems (Second Edition)*. MIT Press, 1992.

[Holliday et al., 2003] JoAnne Holliday, Robert Christian Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1218–1238, 2003.

[Hooper, 2010] Emmanuel Hooper. Strategic and intelligent Smart Grid systems engineering. In *Proceedings of Internet Technology and Secured Transactions*, pages 1–6, 2010.

[IBM-Software, 2012] IBM-Software. Managing Big Data for Smart Grids and smart meters. White paper, IBM Corporation, May 2012.

[IEC, 2003] IEC. Communication networks and systems in substations. International Engineering Consortium IEC 61850-5, 2003.

[IEC, 2007a] IEC. Information security for power system control operations. International Engineering Consortium IEC 62351 Parts 1-8, may 2007a.

[IEC, 2007b] IEC. Parallel redundancy protocol and high-availability seamless redundancy. International Engineering Consortium IEC 62439-3, may 2007b.

[IEEE, 2004] IEEE. IEEE standard communication delivery time performance requirements for electric power substation automation. Institute of Electrical and Electronics Engineers 1646, 2004.

[IEEE, 2011] IEEE. Shortest path bridging. Institute of Electrical and Electronics Engineers IEEE 802.1aq. Draft 4.0, 2011.

[Indelicato, 2008] Max Indelicato. Scalability strategies primer: Database sharding. In *Max Indelicato's blog on distributed, scalable, software systems design and development*. Web. Retrieved June 2012. http://bit.ly/sharding_primer, 2008.

[Informa, 2010] Informa. Informa D&B. In *Quiénes somos – Informa D&B* . Web. Retrieved February 2011. http://www.informa.es/es/quienes-somos, 2010.

[Inmon, 1981] William Inmon. *Effective data base design*. Prentice Hall PTR, 1981. ISBN 0132414899.

[INTEGRIS, 2011] INTEGRIS. INTEGRIS FP7 project intelligent electrical grid sensor communications. In *ICT-Energy-2009 call (number 247938)* http://fp7integris.eu, 2011.

[Jacobs, 2009] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009. ISSN 0001-0782.

[Jacobs and Aulbach, 2007] Dean Jacobs and Stefan Aulbach. Ruminations on multi-tenant databases. In *Proceedings of the 12th Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme*, pages 514–521, 2007. ISBN 978-3-88579-197-3.

[Jiménez-Peris et al., 2002] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 477–484, 2002.

[Jiménez-Peris et al., 2003] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003.

[Johnson et al., 2012] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *The Very Large Data Bases Journal*, 21(2):239–263, 2012.

[Jones et al., 2010] Evan Jones, Daniel Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 603–614, 2010.

[Kapelko and Rialp-Criado, 2009] Magdalena Kapelko and Josep Rialp-Criado. Efficiency of the textile and clothing industry in Poland and Spain. *Fibres & Textiles in Eastern Europe*, 17(3):7–10, 2009.

[Kapitza et al., 2012] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[Kaushik et al., 2010] Rini Kaushik, Milind Bhandarkar, and Klara Nahrstedt. Evaluation and analysis of GreenHDFS: A self-adaptive, energy-conserving variant of the Hadoop Distributed File System. In *Proceedings of CloudCom, 2nd International Conference on Cloud Computing*, pages 274–287, 2010.

[Kemme, 2000] Bettina Kemme. *Database replication for clusters of workstations*. PhD thesis, Department of Computer Science, Swiss Federal Institute of Technology Zurich, 2000.

[Kemme and Alonso, 1998] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 156–163. IEEE Computer Society, 1998.

[Kemme and Alonso, 2000a] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, 2000a.

[Kemme and Alonso, 2000b] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000b.

[Kemme et al., 2001] Bettina Kemme, Alberto Bartoli, and Özalp Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 117–130. IEEE Computer Society, 2001.

[Kendall, 1953] David George Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953. ISSN 00034851.

[Kent and Seo, 2005] Stephen Kent and Karen Seo. Security architecture for the internet protocol. Request for Comments 4301, 2005.

[Khan et al., 2011] Muhammad Zahid Khan, Bob Askwith, Faycal Bouhafs, and Muhammad Asim. Limitations of simulation tools for large-scale wireless sensor networks. In *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications Workshops*, pages 820–825, 2011.

[Kostic et al., 2005] Tatjana Kostic, Otto Preiss, and Christian Frei. Understanding and using the IEC 61850: A case for meta-modelling. *Computer Standards & Interfaces*, 27 (6):679–695, 2005. ISSN 0920-5489.

[Kraska et al., 2009a] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the Very Large Data Bases Endowment*, 2(1):253–264, 2009a.

[Kraska et al., 2009b] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the Very Large Data Bases Endowment*, 2(1):253–264, 2009b.

[Krikellas et al., 2010] Konstantinos Krikellas, Sameh Elnikety, Zografoula Vagena, and Orion Hodson. Strongly consistent replication for a bargain. In *Proceedings of the 26th International Conference on Data Engineering*, pages 52–63, 2010. ISBN 978-1-4244-5444-0.

[Kung and Robinson, 1979] Hsiang-Tsung Kung and John Robinson. On optimistic methods for concurrency control. In *Proceedings of the 5th International Conference on Very Large Data Bases*, page 351. IEEE Computer Society, 1979.

[Kung and Robinson, 1981] Hsiang-Tsung Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[Ladin et al., 1991] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services (extended abstract). *Operating Systems Review*, 25(1):49–55, 1991.

[Lakshman and Malik, 2010] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2): 35–40, 2010.

[Lamport, 1978a] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978a.

[Lamport, 1978b] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978b.

[Lamport, 1998] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lamport, 2006] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[Levandoski et al., 2011] Justin Levandoski, David Lomet, Mohamed Mokbel, and Kevin Zhao. Deuteronomy: Transaction support for cloud data. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 123–133, 2011.

[Lin, 1989] Kwei-Jay Lin. Consistency issues in real-time database systems. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Software Track*, volume 2, pages 654–661, January 1989.

[Lin et al., 2005] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2005. ISBN 1-59593-060-4.

[Lin et al., 2009] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and José Enrique Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Transactions on Database Systems*, 34(2), 2009.

[Liu, 2010] Edwin Liu. Analytics and information integration for Smart Grid applications. In *Proceedings of the 2010 IEEE Power and Energy Society General Meeting*, pages 1–3, 2010.

[Lomet et al., 2012] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE 28th International Conference on Data Engineering*, pages 714–725, 2012. ISBN 978-0-7695-4747-3.

[Lu et al., 2008] Jun-Li Lu, Li-Zhen Wang, Jun-Jia Lu, and Qiu-Yue Sun. Research and application on knn method based on cluster before classification. In *Machine Learning and Cybernetics, 2008 International Conference on*, volume 1, pages 307–313, July 2008.

[Lynch, 2008] Clifford Lynch. Big data: How do your data grow? *Nature*, 455(7209): 28–29, September 2008. ISSN 0028-0836.

[Maia et al., 2010] Francisco Maia, José Enrique Armendáriz-Iñigo, María Idoia Ruiz-Fuertes, and Rui Oliveira. Scalable transactions in the cloud: Partitioning revisited. In *Proceedings of the 12th International Symposium on Distributed Objects, Middleware and Applications*, pages 785–797, 2010. ISBN 978-3-642-16948-9.

[Mak, 2010] Sioe Tho Mak. Knowledge based architecture serving as a rigid framework for Smart Grid applications. In *Proceedings of the IEEE PES Conference on Innovative Smart Grid Technologies Europe*, pages 1–8, 2010.

[Marandi et al., 2010] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 527–536, 2010.

[Martínez-Campillo and Gago, 2009] Almudena Martínez-Campillo and Roberto Fernández Gago. What factors determine the decision to diversify? the case of spanish firms (1997-2001). *Investigaciones Europeas de Dirección y Economía de la Empresa*, 15(1): 15–28, 2009.

[Masdar, 2011] Masdar. Masdar website. In *The reality of future energy*. Web. Retrieved February 2011. www.masdar.ae/en/home/index.aspx, 2011.

[Metke and Ekl, 2010] Anthony Metke and Randy Ekl. Security technology for Smart Grid networks. *IEEE Transactions on Smart Grids*, 1(1):99–107, 2010.

[Monti and Ponci, 2010] Antonello Monti and Ferdinanda Ponci. Power grids of the future: Why smart means complex. In *Proceedings of the IEEE Complexity in Engineering*, pages 7–11, 2010. ISBN 978-0-7695-3974-4.

[Morales-Ortigosa et al., 2009] Sebastián Morales-Ortigosa, Albert Orriols-Puig, and Esther Bernadó-Mansilla. Analysis and improvement of the genetic discovery component of XCS. *International Journal of Hybrid and Intelligent Systems*, pages 81–95, 2009.

[Mosberger, 1993] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[mThink, 2011] mThink. The Smart Grid in Malta. In *Shaping a new era in energy*. Web. Retrieved February 2011. mthink.com/utilities/utilities/smart-grid-malta, 2011.

[Muñoz-Escoí et al., 2006] Francesc Daniel Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 401–410, 2006. ISBN 0-7695-2677-2.

[Muñoz-Escoí et al., 2007] Francesc Daniel Muñoz-Escoí, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Database replication approaches. Technical Report TR-ITI-ITE-07/19, Instituto Tecnológico de Informática, October 2007.

[Nagle, 1987] John Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, apr 1987. ISSN 0090-6778.

[Nambiar et al., 2010] Raghunath Othayoth Nambiar, Nicholas Wakou, Forrest Carman, and Michael Majdalany. Transaction processing performance council (TPC): State of the council 2010. In *Proceedings of the 2nd TPC Technology Conference. Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 1–9, 2010. ISBN 978-3-642-18205-1.

[Navarro et al., 2013] Joan Navarro, Andreu Sancho-Asensio, Agustín Zaballos, Virginia Jiménez-Ruano, David Vernet, and José Enrique Armendáriz-Iñigo. The management system of INTEGRIS: Extending the Smart Grid to the Web of Energy. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science*, pages 119–122. SciTePress, 2013.

[Núñez et al., 2007] Marlon Núñez, Raúl Fidalgo, and Rafael Morales. Learning in environments with unknown dynamics: Towards more robust concept learners. *Machine Learning Research, Journal of*, 8:2595–2628, December 2007. ISSN 1532-4435.

[Orriols-Puig et al., 2010] Albert Orriols-Puig, Xavier Llorà, and David Edward Goldberg. How XCS deals with rarities in domains with continuous attributes. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, pages 1023–1030. ACM, 2010. ISBN 978-1-4503-0072-8.

[Özsu and Valduriez, 1999] Mehmet Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems (2nd edition)*. Prentice-Hall, 1999.

[Palankar et al., 2008] Mayur Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: A viable solution? In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, pages 55–64. ACM, 2008. ISBN 978-1-60558-154-5.

[Pandis et al., 2011a] Ippokratis Pandis, Pinar Tözün, Miguel Branco, Dimitris Karampinas, Danica Porobic, Ryan Johnson, and Anastasia Ailamaki. A data-oriented transaction execution engine and supporting tools. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 1237–1240, 2011a. ISBN 978-1-4503-0661-4.

[Pandis et al., 2011b] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page latch-free shared-everything OLTP. *Proceedings of the Very Large Data Bases Endowment*, 4(10):610–621, 2011b.

[Patiño-Martínez et al., 2005] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[Pavlo et al., 2012] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[Paz et al., 2010a] Alberto Paz, Francisco Pérez-Sorrosal, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Scalability evaluation of the replication support of JOnAS, an industrial J2EE application server. In *Proceedings of the 2010 European Dependable Computing Conference*, pages 55–60. IEEE-CS, 2010a.

[Paz et al., 2010b] Alberto Paz, Francisco Perez-Sorrosal, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Scalability evaluation of the replication support of jOnAS, an industrial J2EE application server. In *Proceedings of the 8th European Dependable Computing Conference*, pages 55–60, 2010b. ISBN 978-0-7695-4007-8.

[Pedone, 1999] Fernando Pedone. *The database state machine and group communication issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[Pedone et al., 1998] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of the 4th Euro-Par '98 Parallel Processing*, pages 513–520, 1998. ISBN 3-540-64952-2.

[Pedone et al., 2000] Fernando Pedone, Matthias Wiesmann, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 464–474, 2000.

[Pedone et al., 2011] Fernando Pedone, Nicolas Schiper, and José Enrique Armendáriz-Iñigo. Byzantine fault-tolerant deferred update replication. In *5th Latin-American Symposium on Dependable Computing*, pages 7–16, 2011.

[Pedone and Oliveira, 2009] Fernnando Pedone and Rui Oliveira. From object replication to database replication. In *Proceedings of the 4th Latin-American Symposium on Dependable Computing, LADC 2009*, page 135. IEEE Computer Society, 2009. doi: 10.1109/LADC.2009.28.

[Peluso et al., 2012] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th ACM/IFIP/USENIX International Middleware Conference*, pages 456–475, 2012. ISBN 978-3-642-35169-3.

[Plattner, 2006] Christian Plattner. *Ganymed: A platform for database replication*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2006.

[Plattner et al., 2006] Christian Plattner, Andreas Wapf, and Gustavo Alonso. Searching in time. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 754–756, 2006. ISBN 1-59593-256-9.

[Plattner et al., 2008] Christian Plattner, Gustavo Alonso, and Mehmet Tamer Özsu. Extending DBMSs with satellite databases. *The Very Large Data Bases Journal*, 17(4): 657–682, 2008.

[Porobic et al., 2014] Danica Porobic, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *Proceedings of the IEEE 30th International Conference on Data Engineering*, pages 688–699, 2014. ISBN 978-1-4799-3480-5.

[Ports et al., 2010] Dan Ports, Austin Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–15, 2010.

[PostgreSQL, 2012] PostgreSQL. The PostgreSQL Global Development Group. In *PostgreSQL 8.4 documentation*. Web. Retrieved July 2013. http://www.postgresql.org, 2012.

[Pothamsetty and Malik, 2009] Venkat Pothamsetty and Sharad Malik. Smart Grid leveraging intelligent communications to transform the power infrastructure. Technical report, Cisco Report, 2009.

[Preen, 2009] Richard Preen. An XCS approach to forecasting financial time series. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2625–2632, 2009. ISBN 978-1-60558-505-5.

[Preguiça et al., 2010] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *Computing Research Repository*, abs/1011.5808, 2010.

[Ramamritham and Chrysanthis, 1996] Krithi Ramamritham and Panos Kypros Chrysanthis. A taxonomy of correctness criteria in database applications. *The Very Large Data Bases Journal*, 5(1):85–97, 1996.

[Repo et al., 2011] Sami Repo, Davide Della Giustina, Guillermo Ravera, Lucio Cremaschini, Stefano Zanini, Josep Maria Selga, and Pertti Járventausta. Use case analysis of real-time low voltage network management. In *Proceedings of the 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies*, pages 1–8, 2011.

[Retolaza and San-Jose, 2008] Jose Luis Retolaza and Leire San-Jose. Efficiency in work insertion social enterprises: A DEA analysis. In *Universidad, sociedad y mercados globales*, pages 55–64, 2008. ISBN 978-1-60558-154-5.

[Rodrigues and Raynal, 2000] Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 288–295, 2000.

[Ryu and Thomasian, 1990] In Kyung Ryu and Alexander Thomasian. Analysis of database performance with dynamic locking. *Journal of the ACM*, 37(3):491–523, 1990.

[Sancho-Asensio et al., 2014] Andreu Sancho-Asensio, Joan Navarro, Itziar Arrieta-Salinas, José Enrique Armendáriz-Iñigo, Virginia Jiménez-Ruano, Agustín Zaballos, and Elisabet Golobardes. Improving data partition schemes in Smart Grids via clustering data streams. *Expert Systems with Applications*, 41(13):5832–5842, 2014. ISSN 0957-4174.

[Schiper, 2006] André Schiper. Dynamic group communication. *Distributed Computing*, 18(5):359–374, 2006.

[Schiper and Raynal, 1996] André Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, 1996.

[Schroeder et al., 2006] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *3rd Symposium on Networked Systems Design and Implementation*, 2006.

[Sciascia et al., 2010] Daniele Sciascia, Marco Primi, Parisa Jalili Marandi, Nicolas Schiper, and Fernando Pedone. Tapioca: Flexible data storage for datacenter applications. Technical Report 2010/04, University of Lugano, May 2010.

[Serrano et al., 2007] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting database replication scalability through partial replication and 1-Copy-Snapshot-Isolation. In *Proceedings of the 13th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 290–297, 2007.

[Shafer et al., 2010] Jeffrey Shafer, Scott Rixner, and Alan Cox. The Hadoop Distributed Filesystem: Balancing portability and performance. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 122–133, 2010.

[Shute et al., 2012] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1: The fault-tolerant distributed RDBMS supporting Google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778, 2012. ISBN 978-1-4503-1247-9.

[Sivasubramanian, 2012] Swaminathan Sivasubramanian. Amazon DynamoDB: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[Stanton, 2005] Jonathan Stanton. The Spread Toolkit. In *Documentation and Resources*. Web. Retrieved July 2013. `http://www.spread.org`, 2005.

[Steinke and Nutt, 2004] Robert Christian Steinke and Gary James Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.

[Stonebraker, 2010] Michael Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010.

[Stonebraker and Weisberg, 2013] Michael Stonebraker and Ariel Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[Stonebraker et al., 2007] Michael Stonebraker, Samuel Madden, Daniel Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1150–1160, 2007.

[Stonebraker et al., 2010] Michael Stonebraker, Daniel Abadi, David DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: Friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[Sutra and Shapiro, 2008] Pierre Sutra and Marc Shapiro. Fault-tolerant partial replication in large-scale database systems. In *Proceedings of the 14th International Euro-Par – Parallel Processing Conference 2008*, pages 404–413. Springer, 2008. ISBN 978-3-540-85450-0.

[Tanenbaum and Steen, 2006] Andrew Stuart Tanenbaum and Maarten van Steen. *Distributed systems: Principles and paradigms (2nd edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

[Thomas, 1979] Robert Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[Thomasian, 1998a] Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998a.

[Thomasian, 1998b] Alexander Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, 1998b.

[Thomasian, 1998c] Alexander Thomasian. Performance analysis of locking methods with limited wait depth. *Performance Evaluation*, 34(2):69–89, 1998c.

[Thusoo et al., 2009] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive – A warehousing solution over a map-reduce framework. *Proceedings of the Very Large Data Bases Endowment*, 2(2):1626–1629, 2009.

[Touch and Perlman, 2009] Joseph Touch and Radia Perlman. Transparent interconnection of lots of links (TRILL): Problem and applicability statement. Request for Comments 5556, may 2009.

[Tözün et al., 2012] Pinar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. *Proceedings of the Very Large Data Bases Endowment*, pages 1–25, 2012. ISSN 1066-8888.

[Traiger et al., 1979] Irving Traiger, Jim Gray, Cesare Galtieri, and Bruce Lindsay. Transactions and consistency in distributed database systems. *IBM Research Report*, RJ2555, 1979.

[Trushkowsky et al., 2011] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael Franklin, Michael Jordan, and David Patterson. The SCADS Director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 163–176, 2011. ISBN 978-1-931971-82-9.

[Vallejo et al., 2012] Alex Vallejo, Agustín Zaballos, Josep Maria Selga, and Jordi Dalmau. Next-generation QoS control architectures for distribution Smart Grid communication networks. *IEEE Communications Magazine*, 50(5):128–134, 2012.

[van Renesse and Guerraoui, 2010] Robbert van Renesse and Rachid Guerraoui. Replication techniques for availability. In *Replication: Theory and Practice*, pages 19–40, 2010. ISBN 978-3-642-11293-5.

[van Renesse and Schneider, 2004] Robbert van Renesse and Fred Barry Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 91–104, 2004.

[van Steen and Pierre, 2010] Maarten van Steen and Guillaume Pierre. Replicating for performance: Case studies. In *Replication*, pages 73–89, 2010. ISBN 978-3-642-11293-5.

[Venayagamoorthy, 2011] Ganesh Kumar Venayagamoorthy. Innovative Smart Grid control technologies. In *Proceedings of the 2011 Power and Energy Society General Meeting*, pages 1–5, 2011.

[Vilaça et al., 2009] Ricardo Manuel Pereira Vilaça, José Orlando Pereira, Rui Carlos Oliveira, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. On the cost of database clusters reconfiguration. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, pages 259–267, 2009.

[Vo et al., 2010] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *Proceedings of the Very Large Data Bases Endowment*, 3(1):506–517, 2010.

[Vogels, 2009] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1): 40–44, 2009.

[Wei et al., 2012] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, 5(4):525–539, 2012.

[White, 2011] Tom White. *Hadoop – The definitive guide: Storage and analysis at Internet scale (2nd edition)*. O'Reilly, 2011. ISBN 978-1-449-38973-4.

[Wiesmann and Schiper, 2005] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.

[Wiesmann et al., 2000] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–215, 2000.

[Wilson, 1995] Stewart Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[Wilson, 1999] Stewart Wilson. Get real! XCS with continuous-valued inputs. In *Learning Classifier Systems, From Foundations to Applications*, pages 209–222, 1999. ISBN 3-540-67729-1.

[Wong and Schulenburg, 2007] Sor Ying (Byron) Wong and Sonia Schulenburg. Portfolio allocation using XCS experts in technical analysis, market conditions and options market. In *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late breaking papers*, pages 2965–2972, 2007. ISBN 978-1-59593-698-1.

[Xu and Ii, 2005] Rui Xu and Ii. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, May 2005.

[Yang et al., 2011] Qiang Yang, Javier Barria, and Tim Green. Communication infrastructures for distributed control of power distribution networks. *IEEE Transactions on Industrial Informatics*, 7(2):316–327, 2011.

[Yichi Zhang et al., 2011] Lingfeng Wang Yichi Zhang, Weiqing Sun, Robert Green II, and Mansoor Alam. Artificial immune system based intrusion detection in a distributed hierarchical network architecture of Smart Grid. In *Proceedings of the 2011 IEEE Power and Energy Society General Meeting*, pages 1–8, 2011.

[Zaballos et al., 2010] Agustín Zaballos, Alex Vallejo, Guillermo Ravera, and Josep Maria Selga. Simulation and analysis of a QoS multipath routing protocol for smart electricity networks. *IARIA International Journal on Advances in Networks and Services Transactions on Smart Grids*, 3(3-4), 2010.

[Zaballos et al., 2011] Agustín Zaballos, Alex Vallejo, and Josep Maria Selga. Heterogeneous communication architecture for the Smart Grid. *IEEE Network*, 25(5):30–37, 2011.