UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Autonomic Replicated Software Transactional Memory

Maria Isabel Catarino Couceiro

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisor: Doctor Paolo Romano

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering
Jury final classification: Pass with Merit

Jury

2015

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Autonomic Replicated Software Transactional Memory

Maria Isabel Catarino Couceiro

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisor: Doctor Paolo Romano

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering
Jury final classification: Pass with Merit

Jury

Funding Institutions

Fundação para a Ciência e Tecnologia

2015

Agradecimentos

Começo por agradecer aos meus orientadores, Luís Rodrigues e Paolo Romano. Obrigado por acreditarem em mim, pela oportunidade de trabalhar convosco e por todo o vosso inesgotável apoio.

Quero também agradecer aos meus colegas no Grupo de Sistemas Distribuídos, em especial ao meu co-autor Pedro Ruivo, pela sua contribuição importante neste trabalho.

E, por último, agradeço à minha família e amigos por todo o apoio, carinho, incentivo e paciência ao longo desta viagem, em especial ao João e aos meus pais, que estiveram sempre presentes.

Finalmente, quero agradecer ao Instituto Superior Técnico, ao INESC-ID e ao Grupo de Sistemas Distribuídos pelas instalações e acolhimento. Este trabalho foi parcialmente suportado pela Fundação para a Ciência e Tecnologia, através da bolsa de Doutoramento (SFRH/BD/81685/2011), pelos projectos Pastramy (PTDC/EIA/72405/2006), Aristos (PTDC/EIA- EIA/102496/2008), e ADAAS (CMUPT/ELE/0030/2009). Este trabalho foi ainda parcialmente suportado pela Comissão Europeia, através do projecto STReP Cloud-TM (contracto número 257784, do sétimo programa quadro).

Lisboa, Julho 2015

Maria Couceiro

Para os meus pais,

António e Margarida.

Resumo

Os sistemas de memória transacional em software (STM) emergiram como um paradigma poderoso para desenvolver aplicações concorrentes. Ao evitar que o programador lide explicitamente com mecanismos de concorrência de nível baixo, as STMs aumentam a fiabilidade do código e diminuem o tempo de desenvolvimento.

Por razões de escalabilidade e tolerância a falhas, é importante construir concretizações distribuídas deste paradigma que permitam o uso de sistemas transacionais em ambientes como a nuvem. A replicação tem claramente um papel importante nestas plataformas, já que é um mecanismo chave para garantir a durabilidade dos dados face às inevitáveis falhas de nós. No entanto, há inúmeras técnicas para manter a consistência dos dados replicados, cada uma delas com um desempenho diferente de acordo com o padrão de carga da aplicação e não existe uma técnica única que consiga atingir um desempenho óptimo em todo e qualquer cenário. Para além disso, devido à crescente complexidade destes sistemas e imprevisibilidade dos padrões de carga das aplicações, a gestão manual torna-se uma tarefa complexa, fatigante e propensa a erros.

Para contornar estas questões, esta tese investiga uma arquitetura autónoma para a replicação de STMs que suporte a mudança dos protocolos de replicação de acordo com os padrões de carga, de forma a atingir o melhor desempenho possível, usando técnicas que não requeiram intervenção humana. Primeiro, estuda o uso de técnicas de aprendizagem de máquina para suportar a gestão autónoma dos protocolos de replicação. Depois, a tese apresenta uma solução que tem por base duas abordagens de aprendizagem de máquina para guiar a adaptação numa STM distribuída, que tem a capacidade de trocar entre protocolos de replicação baseados em certificação. Por último, propõe uma moldura de execução para STMs distribuídas autónomas que suporta múltiplos protocolos de replicação e dois mecanismos de troca: uma que obriga o sistema a parar de processar transações enquanto a troca de protocolos é feita, e outra que permite que o programador defina como os dois protocolos podem co-existir durante o período de transição.

Abstract

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications. By sparing the programmer from the burden of explicitly dealing with low-level concurrency mechanisms, STMs increase the code reliability and shorten the development time.

For scalability and fault-tolerance reasons, it is relevant to build distributed implementations of this paradigm allowing the deployment of transactional systems in environments such as the cloud. Replication clearly plays a role of paramount importance in these platforms, as it represents the key mechanism to ensure data durability in face of unavoidable node failures. However, there is a myriad of techniques to maintain the consistency of the replicated data, each performing differently according to the workload and there is no "one-size fits all" technique that achieves optimal performance in any scenario. In addition, due to the growing complexity of these systems and unpredictability of the workloads, manual management becomes a complex, tiresome and error-prone task.

To circumvent these issues, this thesis investigates an autonomic architecture for the replication of STMs that supports the change of its replica consistency protocol according to the workload, in order to provide the best throughput possible, using techniques that do not require human intervention. First, it studies the use of machine learning to support the autonomic management of the replica consistency protocols. Then, the thesis presents a solution that relies on two machine learning approaches to drive the adaptation in a distributed STM that is able to switch between certification-based protocols. Finally, it proposes a framework for autonomic distributed STMs that supports multiple replica consistency protocols and two switching mechanisms: one forcing the system to stop processing transactions while the protocol switch is in progress, and another which allows the programmer to define how the two switching protocols can co-exist during the transition period.

Palavras Chave
Keywords

Palavras Chave

Sistemas Distribuídos

Memória Transacional em Software

Replicação de Dados

Coerência de Dados

Sistemas Adaptativos

Gestão Autónoma

Aprendizagem de Máquina

Processamento de Transações Distribuídas

Tolerância a Falhas

Keywords

Distributed Systems

Software Transactional Memory

Data Replication

Data Consistency

Adaptive Systems

Autonomic Management

Machine Learning

Distributed Transaction Processing

Fault Tolerance

Index

List of Figures

List of Tables

1 Introduction

Transactional Memory (TM) can be defined as a generic non-blocking synchronization construct that allows correct sequential objects to be converted automatically into correct concurrent objects (Marathe & Scott 2004) and borrows the notions of atomicity, consistency, and isolation from database transactions. The abstraction relies on an underlying runtime system that is able to enforce some target consistency criteria for the concurrent execution (such as serializability (Bernstein, Shipman, & Wong 1979)). If the consistency criteria is met, the transaction is committed, otherwise its execution is aborted and the transaction is restarted. The ability to abort transactions eliminates the complexity and potential deadlocks of fine-grain locking protocols. Executing non-conflicting transactions simultaneously creates the opportunity to achieve high performance by yielding speed-ups with respect to sequentially executed code.

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications (Herlihy, Luchangco, Moir, & Scherer 2003; Shavit & Touitou 1997; Dice, Shalev, & Shavit 2006). When using STMs, the programmer is not required to deal explicitly with thread synchronization mechanisms. Instead, she has only to identify the sequence of instructions, or transactions, that need to access and modify concurrent objects atomically. As a result, code reliability increases and software development time is shortened.

For scalability and fault-tolerance reasons, it is relevant to build distributed implementations of this paradigm allowing the deployment of transactional systems in environments such as the cloud (Romano, Rodrigues, Carvalho, & Cachopo 2010). Replication clearly plays a role of paramount importance in these platforms, as it represents the key mechanism to ensure data durability in face of unavoidable node failures. In replicated STM implementations, as the name suggests,the STM is replicated in multiple nodes, which have to coordinate to ensure the consistency of the cached data. These replication protocols take inspiration from the vast literature on replication of database systems (Kemme & Alonso 1998; Pedone, Guerraoui, & Schiper 2003; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000), but also, more recently, TM systems (Carvalho, Romano, & Rodrigues 2010; Couceiro, Romano, Carvalho, & Rodrigues 2009).

1.1 Problem Statement

It has been shown that different protocols can produce the best results for different workloads (Couceiro 2009; Carvalho, Romano, & Rodrigues 2011a), and real applications exhibit very heterogeneous workload patterns during their execution. So, it is not trivial to match the correct protocol with the current workload: there is no "one-size-fits-all" configuration that achieves the optimal performance for all workloads or deployment scenarios. Additionally, the complexity of these systems, multiple deployment configurations and unpredictability of the workloads result in a myriad of different combinations, turning manual management into a complex, tiresome and error-prone task. Therefore, the problem that this thesis addresses is the following:

Is it possible to design an autonomic replicated STM that outperforms non-adaptive replicated STM solutions, across a wide range of different workload scenario?

To answer this problem, the thesis investigates solutions that leverage on the experience gained in two related fields. First, studying the state of the art of distributed STM systems provides an insight on the key aspects of the architecture and on the main bottlenecks of the system when faced with heterogeneous workloads. Second, this work also leverages on existing results in the area of adaptive (non-distributed) STM, but modifying them to cope with the challenges of a distributed environment.

1.2 Contributions

With this work, I achieved the following contributions:

- The identification of the relevant features that should be considered in the policies that control the adaptation of replicated STM systems.
- The design of different mechanisms aimed to derive automated policies for guiding adaptation.

- The design of protocols that support the switching or even the possible coexistence of multiple replication protocols.

1.3 Results

Given the proposed contributions, the results of this thesis are twofold:

- A prototype of an autonomic replicated STM supporting different adaptation triggering techniques and replication algorithms.
- An extensive evaluation study assessing both the accuracy of the mechanisms to trigger adaptation and the effectiveness of the prototype of the autonomic replicated STM.

1.4 Research History

The main inspiration for this work came from my master thesis (Couceiro 2009), where I implemented four different replication protocols for a distributed STM. From this work experience, it was possible to draw the following conclusions:

1. All protocols responded differently to distinct workloads and there was no single protocol that outperformed the rest for all the workloads tested (e.g.: read dominated, write dominated, etc.).
2. It is not obvious which protocol achieves the highest throughput for which workload and system configuration and, consequently, manual specification of the combinations and management may be incomplete and lead to poor performance, so autonomic techniques of detecting the need to change protocol and identify the best one need to be applied.

Furthermore, the work of Nuno Carvalho (Carvalho 2011) defined a generic framework for replicated STM which supports the implementation of different replication protocols but provides neither means to switch between them nor mechanisms to trigger this switch.

These observations motivated me to perform research on an autonomic architecture for the replication of STMs that supports the change of its replication protocol according to the workload, in order to provide the best throughput possible, using techniques that do not require human intervention.

The first step to build an autonomic replicated STM was to understand which features are relevant to trigger adaptation. From the plethora of metrics that can be collected from an executing system, only a few are pertinent to determine the state of the system. This is strictly connected with the kind of adaptation triggering mechanism used, which is also tested in this stage. These techniques must be able to accurately predict the STMs performance under different configurations in order to correctly estimate the best. This step was materialized in the following publication: *"A Machine Learning Approach to Performance Prediction of Total Order Broadcast Protocols"*, by Maria Couceiro, Paolo Romano and Luís Rodrigues, in Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'10), Budapest, Hungary, 2010.

The following step consisted of implementing a first version of the prototype which is able to switch between a restricted type of replication protocols. Having this purpose in mind, I opt for supporting certification-based protocols, based on the fact that they have a very similar structure, diverging only in the validation phase. This was a logical first step towards building a complex prototype while serving as an effective proof-of-concept. This step was materialized in the following publication: *"PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids"*, by Maria Couceiro, Paolo Romano and Luís Rodrigues, in Proceedings of the ACM/IFIP/USENIX 12th Middleware Conference, Lisbon, Portugal, 2011.

The final step consisted of building a prototype supporting multiple replication protocols, regardless of their structure. This more diverse library of available protocols widens the chances that the system can deliver the best performance possible for any workload generated by the application. This step was materialized in the following publication: *"Chasing the Optimum in Replicated In-memory Transactional Platforms via Protocol Adaptation"*, by Maria Couceiro, Pedro Ruivo, Paolo Romano and Luís Rodrigues, in Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2013), Budapest, Hungary, 2013. The results were also published as a journal paper with the same title in IEEE Transactions on Parallel & Distributed Systems.

It is worth mentioning that this work was integrated in the CloudTM[1] European project, which aimed at developing a Self-Optimizing Distributed Transactional Memory middleware for the cloud. Parts of this work are available for download[2]. In addition, parts of the work presented in this thesis were also published as two book chapters: *“Self-tuning in Distributed Transactional Memory”* , by Maria Couceiro, Diego Didona, Luís Rodrigues, Paolo Romano, Transactional Memory. Foundations, Algorithms, Tools, and Applications, Lecture Notes in Computer Science Volume 8913, 2015, and *“Towards Autonomic Transactional Replication for Cloud Environments”*, by Maria Couceiro, Paolo Romano and Luís Rodrigues, European Research Activities in Cloud Computing, Cambridge Scholars Publishing, 2012.

1.5 Structure of the Document

This document is organized as follows. Chapter 2 introduces fundamental concepts which are relevant for the context of the contributions presented in the thesis. Chapter 3 presents the challenges associated with using machine learning techniques to derive fine-grained performance prediction models of total order broadcast protocols. Chapter 4 introduces PolyCert, a self-optimizing replicated STM that supports the simultaneous use of three certification protocols and relies on machine-learning techniques to determine, on a per transaction basis, the certification strategy to be adopted. Chapter 5 describes MorphR, which is a framework supporting automatic adaptation of the replication protocols employed by in-memory transactional platforms. Finally, Chapter 6 concludes this document.

[1]`http://www.cloudtm.eu`
[2]`http://www.cloudtm.eu/downloads`

2 Background

This chapter starts by briefly describing the main concepts of the four fundamental areas that are on the core of the presented work: distributed and replicated databases, distributed shared memory (DSM), transactional memory, and autonomic computing. Then it presents a survey of the main results in the area of Distributed Transactional Memory systems and adaptation in STM.

2.1 Distributed and Replicated Database Systems

The problem of distributing a TM is naturally closely related to the problem of database (DB) distribution and replication, given that both TMs and DBs share the same key abstraction of atomic transactions. The next paragraphs briefly survey some of the most representative distribution and replication database algorithms that can be found in the literature.

Two-phase commit (2PC) is one of the most relevant protocols for distributed databases. In this protocol, a node plays the role of the coordinator while the other nodes involved in the transaction are the participants. It consists of four stages: i) the coordinator sends a commit request to all the participants; ii) the participants determine their vote and send it to the coordinator; iii) the coordinator then decides on the outcome (abort the transaction if at least one of the participants chose to abort, or commit it otherwise) and sends it to the participants; which iv) apply the write items if the transaction is to be committed, and release the locks.

Due to their simplicity, data replication/distribution solutions based on 2PC are extremely popular especially in industrial products (Oracle 2015; Marchioni 2012; DB2 2015). Unfortunately, they are also well known to suffer from scalability problems due to the rapid growth of the distributed deadlock rate as the number of replicas in the system grows (Gray, Helland, O'Neil, & Shasha 1996). This means that 2PC could be (and very often is, especially in industrial products) used to ensure consistent evolution of replicated databases. On the other hand, a wide range of alternative techniques for database replication have been proposed in literature.

Among the plethora of existing database replication solutions, we can identify two main approaches: primary-backup and Total Order Broadcast (TOB) based solutions.

The primary-backup approach, also called passive replication (Budhiraja, Marzullo, Schneider, & Toueg 1993), is based on the idea of processing all transactions that update the state of the database exclusively at a single node (also called master or primary), whereas the remaining replicas can only run read-only transactions. Upon failure of the master, a backup replica is elected to become the new master.

The fulcrum of modern database replication schemes (Pedone, Guerraoui, & Schiper 2003; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000; Cecchet, Marguerite, & Zwaenepole 2004) is the reliance on a Total Order Broadcast primitive (Powell 1996), typically provided by some Group Communication System (GCS) (Amir, Danilov, & Stanton 2000; Miranda, Pinto, & Rodrigues 2001). TOB plays a key role to enforce, in a non-blocking manner, a global transaction serialization order without incurring in the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commit protocols, which require much finer grained coordination and fall prey of deadlocks (Gray, Helland, O'Neil, & Shasha 1996). On the other hand, TOB is a costly communication primitive, as enforcing agreement (in a non-blocking way) on message delivery order requires the exchange of multiple message rounds among nodes (Lamport 1998).

Existing TOB-based database replication literature can be coarsely classified in two main categories, depending on whether transactions are executed optimistically (Kemme & Alonso 1998; Pedone, Guerraoui, & Schiper 2003; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000) or conservatively (Schneider 1993). Starting with the latter approach, which is also known as state-machine replication, all nodes execute the same set of transactions in the same order; the transactions are shipped to the nodes through TOB and, consequently, all nodes receive and execute transactions in the same order. This implies that i) transactions are executed serially at each node, and ii) the code executed by the transaction is fully deterministic, which may not be the case for transactions with conditions depending on a random number generator or the "date" function, for instance. On the other hand, optimistic solutions tackle this problem by locally processing transactions on a single node and by validating the results at the end of their execution through a TOB-based certification procedure aimed at detecting remote conflicts between concurrent transactions. This solves any scheduling differences and other non-deterministic issues among nodes by executing the transaction in only one node and relying on a deterministic validation, executed in the same order in all replicas, to ensure system-wide consistency.

One of the most striking differences between DBs and DSTMs is the transaction execution time, which is several orders of magnitude slower in the former when compared to the latter (Romano, Carvalho, & Rodrigues 2008). This fact amplifies the impact on performance of the coordination phase among the nodes, which is the most expensive phase in a DSTM transaction and stresses the need for replica consistency schemes tailored for DSTMs (Palmieri, Quaglia, & Romano 2010; Carvalho, Romano, & Rodrigues 2010).

2.2 Distributed Shared Memory

Distributed Shared Memory systems (Li & Hudak 1986; Keleher, Cox, Dwarkadas, & Zwaenepoel 1994; Bershad, Zekauskas, & Sawdon 1993) aim at combining the advantages of two architectures: shared memory systems (all processors have equal access to a single global physical memory) and distributed memory systems (multiple processing nodes that communicate by means of message passing). A DSM system implements a shared memory abstraction on top of message passing distributed memory systems. It inherits the ease of programming and portability from the shared memory programming paradigm and the cost-effectiveness and scalability from distributed memory systems (Protic, Tomasevic, & Milutinovic 1996; Nitzberg & Lo 1991).

The expected behaviour of the memory system is defined by a memory consistency model. To overcome the strong performance overheads introduced by straightforward DSM implementations ensuring strong consistency guarantees (Lamport 1979) with the granularity of a single memory access, several DSM systems were developed to achieve better performance through relaxing memory consistency guarantees (Keleher, Cox, & Zwaenepoel 1992). Unfortunately, developing software for relaxed consistency models can be challenging as programmers are required to fully understand sometimes complicated consistency properties to maximize performances without endangering correctness.

There are two main approaches to locate the data that is distributed by the nodes in the system: broadcast-based and directory-based algorithms (Li & Hudak 1989). In the first approach, no information about the location of data items is maintained: whenever a node wants to access a data item, it broadcasts a message that is replied to by whoever is in the possession of the item. In directory-based algorithms, the information about the current location of shared data is kept in a global directory. Later in the text we will see that DSTMs further developed this mechanisms to adapt to transactional environments to guarantee a degree of efficiency in locating and fetching remote data items that these first implementations were unable to offer in such scenarios.

Although these systems provide the abstraction of a global shared address space in a distributed system, most of them were not integrated with transactional mechanisms (with a few exceptions (Shrivastava, Dixon, & Parrington 1991)). This puts barriers to the fast development of correct concurrent applications, requiring programmers to implement lock-based concurrency control schemes to regulate concurrent accesses to shared objects/memory regions.

2.3 Transactional Memory

The transactional memory abstraction can be implemented with hardware support or entirely on software. An hardware implementation of transactional memory (Herlihy & Moss 1993) is based on extensions to multiprocessor cache consistency protocols and provides support for a flexible transactional language for writing synchronization operations, which can be written as a transaction. Recently, IBM (Wang, Gaudet, Wu, Amaral, Ohmacht, Barton, Silvera, & Michael 2012) and Intel (Yoo, Hughes, Lai, & Rajwar 2013) have inserted hardware support for transactional memory in their last processors, which gave a renewed impulse to research in this area. There are also systems combining both software and hardware implementations, known as Hybrid Transactional Memory (Damron, Fedorova, Lev, Luchangco, Moir, & Nussbaum 2006; Minh, Trautmann, Chung, McDonald, Bronson, Casper, Kozyrakis, & Olukotun 2007).

The first STM was proposed in Shavit & Touitou (1997). The interest in the area was spurred again in 2003 by a couple of influential papers (Herlihy, Luchangco, Moir, & Scherer 2003; Harris & Fraser 2003) and by the advent of multicore chips. Since then, research on TMs has addressed a wide range of complementary aspects including hardware and operating systems (OSs) support (Ramadan, Rossbach, Porter, Hofmann, Bhandari, & Witchel 2007), language integration (Harris & Fraser 2003; Shpeisman, Adl-Tabatabai, Geva, Ni, & Welc 2009), as well as algorithms and theoretical foundations (Guerraoui & Kapalka 2008; Gramoli, Harmanci, & Felber 2008).

The first proposed STMs were lock-free, ruling out the possibility to incur in deadlock situations. These solutions (Moir 1997) are typically considerably complex and impose some performance penalties. A second generation of STMs ensures the weaker progress guarantee of obstruction-freedom, guaranteeing progress only in the case of a thread that is running for itself for long enough. As a key benefit, obstruction-free STMs are easier to implement efficiently (Herlihy, Luchangco, Moir, & Scherer 2003). A third approach is lock-based STM (Felber, Fetzer, & Riegel 2008), which lacks any progress guarantee (as above) but is able to achieve better performance. Another approach for concurrency control is based on multi-versioning (Cachopo & Rito-Silva 2006; Perelman, Byshevsky, Litmanovich, & Keidar 2011), in which the STM maintains multiple versions of the data items; this approach trades off using more memory for supporting a greater degree of concurrency as is particularly tailored for read-dominated workloads as read-only transactions are sheltered from aborts as the versions they read are never modified. A recent study (Diegues, Romano, & Rodrigues 2014) has evaluated empirically the performances of hardware, software and hybrid implementation of TMs. The results of this study highlight that STMs remain the most competitive approach for applications that generate long running transactions that access a large number of data items.

TMs, independently of whether they are implemented in hardware or software, rely on one component to ensure that the system as a whole makes progress, the contention manager (Guerraoui, Herlihy, & Pochon 2005). When transactions conflict, the contention manager is responsible for deciding which will abort and which continues executing, based on the implemented policies. The work of Scherer III & Scott (2004) surveys the most representative contention managers, comparing them experimentally. One of the authors' main conclusions was that different contention management policies work better for different benchmark applications and that none is able to achieve all-around best results.

2.4 Autonomic Computing

Autonomic Computing (AC) (Horn 2001) refers to building computing systems that can manage themselves to satisfy a set of high-level policies with a minimum of human interven-

Self-Configuration	An AC system configures itself according to high-level goals.
Self-Optimization	An AC system optimizes its use of resources in order to improve performance or quality of service.
Self-Healing	An AC system detects problems and attempts to fix or mitigate their impact.
Self-Protection	An AC system protects itself from malicious attacks and tunes itself to achieve security, privacy, and data protection.

Table 2.1: The Self-* properties of Autonomic Computing.

tion. These systems will continuously monitor their performance and optimize their status autonomously to self-adapt to changing conditions. The main problems it tries to solve include minimizing the effort associated to the maintenance of computing systems with growing complexity and attenuate the expected lack of IT personal to support them.

The four core properties of AC, know as the Self-* properties, are summarized in Table 2.1. These properties enable computing systems to adjust their operation in the face of changing conditions (workload, components or other external factors) and better respond to hardware or software failures (Kephart & Chess 2003) .

Even though the concept of AC was introduced by Horn (2001), the technologies enabling its implementation rely on knowledge from multiple areas, some of which well established, such as software architecture, artificial intelligence, and control theory, while it also draws inspiration from biological systems like ant colonies or insect swarms to introduce autonomic capabilities to the systems (Kephart 2005).

AC has been applied in many diverse contexts, ranging from wireless sensor network routing to data center resource management and grid power management, to name but a few (Huebscher & McCann 2008). Section 2.6 briefly surveys a core research area of AC for this work: adaptive (non-distributed) STMs.

2.5 Distributed Software Transactional Memory

This section presents a survey of the main results in the area of DSTM systems. Each subsection represents a main design choice when building such a system, namely programming model, data model, data distribution, and replication. Each of these categories is further divided into several sub-categories and represents a distinct dimension in a DSTM system that ultimately defines its application.

2.5.1 Programming Model

The programming model of a DSTM encompasses two main design choices: how the transactional data is moved and whether the transaction's operations are fully known in advance or not, which may place restrictions for programmers working with these systems.

2.5.1.1 Data vs. Control Flow

DSTMs can be classified according to two classic distributed programming models (Herlihy & Sun 2007): the data flow model, in which transactions are immobile and data items are migrated to the node executing the transaction, and the control flow model, in which data items are immobile and transactions invoke them through remote procedure calls (RPCs) (Birrell & Nelson 1984). There are also systems that allow the co-existence of both approaches, called hybrid.

In more detail, in the data flow model, the transaction executes in a single node, gathering data items from the other nodes, and the synchronization is delayed until the end of its execution. Then the node performs a local validation and the transaction commits if no other concurrent transaction executed a conflicting access. The contention manager module is responsible for this synchronization (Scherer III & Scott 2004). One of the major strengths of this approach is the absence of a distributed commit protocol, as a transaction can simply commit locally if it reaches the complete phase (and if data does not need to be replicated synchronously for fault-tolerance purposes).

The data flow model is typically implemented using a directory-based approach, in which the last location of an object is saved in a distributed directory such that the costs to locate and move it are bounded. Ballistic (Herlihy & Sun 2007), Relay (Zhang & Ravindran 2009) and Combine (Attiya, Gramoli, & Milani 2010) implement this approach. Ballistic is a location aware protocol for metric space networks in which nodes are organized in hierarchical clusters, whose leaves are the physical nodes; this way, when a transaction requests an object, the request rises in the hierarchy until it finds a downward link, which is then followed until the request reaches the cached copy of the object. Relay is based on a fixed spanning tree and keeps a path vector with the nodes the message traversed until it reaches the requested object, which is then moved to the requesting node using the reverse vector. Finally, contrarily to the previous two protocols, Combine does not assume that the communication links preserve the order of the messages and combines concurrent requests to objects in the same message, which is exchanged by the nodes organized in an overlay tree. Similarly to Ballistic, Quorum-based replication (Zhang & Ravindran 2011) is also based on a overlay tree in which the communication between nodes forms a metric, but additionally it offers fault-tolerance by having multiple replicas coordinating in a quorum to ensure consistency among the copies at commit time.

As previously stated, in the control flow model, the computations move from node to node through RPCs as the data items are statically assigned to the DSTM nodes. In order to access an item, a transaction performs an RPC to the item's home node (i.e., the node responsible storing the item) and waits for the result of the tentative updates. The synchronization is generally provided by two-phase locking together with some form of deadlock detection. As for the commit protocol, this approach typically uses two-phase commit to guarantee that either all nodes commit the transaction or none does.

Snake-DSTM (Saad & Ravindran 2011b) is an example of a control flow DSTM. Any node that wants to read from or write to an object contacts the object's home node though a remote call (using the Remote Method Invocation mechanism), which may trigger other remote calls to different objects and, consequently, these remote accesses form a call graph, which is used in the commit decision process.

Finally, (Bocchino, Adve, & Chamberlain 2008) presents a hybrid solution. The programmer chooses the model according to the characteristics of the transaction with the objective of maximizing the performance of the system: either the data is moved to the node executing the transaction or one or more operations over that data are executed on the node storing it. HyFlow (Saad & Ravindran 2011a) also allows the coexistence of models: each object has an owner that is responsible for handling the requests from other nodes but this ownership may change when the data flow model is used and, in this case, upon the commit of the transaction, the new owner is broadcast in a message.

2.5.1.2 Static vs. Dynamic Transactions

Static transactions require that all the concurrent objects accessed by a transaction are known before it actually starts. This kind of transactions can be defined as a function that takes as arguments the read and write set of the transaction (the data items read and written, respectively), accesses these items deterministically, and atomically determines the new values to be stored (Shavit & Touitou 1997).

In contrast, dynamic transactions (Herlihy, Luchangco, Moir, & Scherer 2003) allow for transactional data items to be created dynamically, as well as for determining the sequence of data items to access and update based on the values observed in data items, instead of having this sequence pre-defined. In this case, the read and write sets are only known when the execution of the transaction ends.

Sinfonia (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007) introduces the minitransaction primitive, which consists of a set of read, write and compare items known in advance. The compare items set controls if the minitransaction commits or aborts, while the read and write set determine the data it returns and updates. By using this primitive, an application can update data in multiple nodes efficiently while ensuring the classic ACID properties.

However, this system is not the norm, as even the first solutions published (Manassiev, Mihailescu, & Amza 2006; Kotselidis, Ansari, Jarvis, Lujan, Kirkham, & Watson 2008; Bocchino, Adve, & Chamberlain 2008) opt for the dynamic approach as the latter is more flexible and simplifies the development of complex applications.

2.5.2 Data Granularity

The granularity of the transactional data items determines the size of the data blocks managed by the consistency protocols.

DSTM systems use mainly three different degrees of granularity for transactional data items, namely word, page and object. While a coarser level of granularity, such as pages, may be more beneficial in terms of performance for applications with a lower degree of concurrency and high data locality, fine-grain data items, such as words, decrease the probability of false conflicts in high concurrency scenarios (a problem referred to in literature as false sharing (Bolosky

& Scott 1993)). However, these two levels of granularity have fixed sizes and disregard the mapping between the data item manipulated by the programmer and the data manipulated by the transactional engine. Object-level granularity, with sizes that vary from a few words to more than a page, builds the bridge between the programming API and the data that is actually considered to validate and commit the transaction.

Sinfonia (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007) and Cluster-STM (Bocchino, Adve, & Chamberlain 2008) are two examples of systems using word-level granularity. Sinfonia sends the words manipulated by the transaction piggybacked in the first phase of its commit protocol (as explained in Section 2.5.3.2) in order to minimize the overhead of sending a (possibly large) collection of words. Cluster-STM employs a different approach by allowing programmers to decide whether they want to move data or execute the computation on the node where it is placed, in order to minimize the transfer of data between nodes.

DMV (Manassiev, Mihailescu, & Amza 2006) implements page-level granularity as it inherits this approach from software implementations of DSM (Protic, Tomasevic, & Milutinovic 1996; Nitzberg & Lo 1991).

DSMTX (Kim, Raman, Liu, Lee, & August 2010), on the other hand, has two granularity levels according to the nature of the access to the data item. For reading data items, the whole page is transferred as this system exploits the locality of the accesses of applications. When transactions update data items, the granularity level in this case are words, as the speculative nature of this transactions often results in having different transactions in different stages of their execution accessing several words in the same page, so this strategy decreases the abort rate.

The most popular granularity level for DSTMs is, probably, the object, as it allows more flexible consistency protocols and programming APIs, while avoiding the problems of false sharing of pages and the overhead of maintaining the metadata of such a small level of granularity as words (Bieniusa & Fuhrmann 2010; Herlihy & Sun 2007; Kotselidis, Ansari, Jarvis, Lujan, Kirkham, & Watson 2008; Carvalho, Romano, & Rodrigues 2010; Couceiro, Romano, Carvalho, & Rodrigues 2009; Palmieri, Quaglia, & Romano 2010; Kotselidis, Lujan, Ansari, Malakasis, Kahn, Kirkham, & Watson 2010; Zhang & Ravindran 2011; Saad & Ravindran 2011b; Ruivo, Couceiro, Romano, & Rodrigues 2011; Zhang & Ravindran 2009; Attiya, Gramoli, & Milani 2010; Dash & Demsky 2011).

2.5.3 Data Distribution

This subsection concerns the issues regarding the distribution of data in transactional memory systems, more specifically where to store and how to locate it, how to ensure all nodes involved in a transaction take the same action and, finally, optimizations to hide the latency of accessing remote objects.

2.5.3.1 Data Placement

When the data is distributed among the several nodes in the system, systems must implement a mechanism to both distribute and locate the data (and possible replicas). Four representative examples are presented in the next paragraphs.

One recent trend for DSTMs are key/value store systems supporting transactions over multiple data items. As the name suggests, these systems enable the applications to store data (usually objects) without any schema associated, simply mapping the keys associated with the data items to nodes in the system, usually with the help of hash functions. Two examples of such systems are Scalaris (Schütt, Schintke, & Reinefeld 2008; Schintke, Reinefeld, Haridi, & Schütt 2010) and Infinispan (Marchioni 2012).

DecentSTM's (Bieniusa & Fuhrmann 2010) algorithm is based in globally accessible objects (GAOs) (Saballus, Eickhold, & Fuhrmann 2008), which are a distributed data structure for shared-memory objects. To place and locate the GAOs in the system, DecentSTM relies on a distributed hash table (DHT) (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001) that is used as an oracle to retrieve the reference to objects.

Similarly, Scalaris relies on what is known as symmetric data replication (Ghodsi, Alima, & Haridi 2007) on top of a structured overlay like Chord# (Schutt, Schintke, & Reinefeld 2006). All key/value pairs are replicated r times in a ring that is subdivided in r equal parts. Infinispan uses consistent hashing (Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997) to place and locate objects and their replicas. This technique allows for locating the nodes responsible of storing keys locally, avoiding extra communication hops. On the other hand, it also imposes maintaining information on full membership of the system at each node (unlike solutions using structured overlays).

In these three systems, the owners of the data items do not change over time, as the ownership of a key depends on the key itself and the ID of a node, which is constant; in normal operation conditions, a key will only change owners if the number of nodes in the platform varies. However, as previously presented, in systems following the data flow model (Herlihy & Sun 2007; Attiya, Gramoli, & Milani 2010; Zhang & Ravindran 2009), distributed directories are employed to maintain the current location of objects in the system, as the ownership of data items changes over time according to which node requests the access to the object.

2.5.3.2 Consistency Protocols

In order to guarantee that all nodes storing data read or written during a transaction agree on its outcome, an atomic agreement process must be executed. While there are systems that only require local commit in the node executing the transaction, others resort to more complex synchronization protocols, such as 2PC or even consensus (Lamport 1998).

In single copy cache consistency data flow systems (Herlihy & Sun 2007; Attiya, Gramoli, & Milani 2010; Zhang & Ravindran 2009), synchronization is performed by transferring the data to the nodes executing the transaction, so no two concurrent transactions will try to commit at the same time (one object will only be updated by one transaction at the time) and a simple local commit protocol will suffice.

Several systems employ two-phase commit schemes. Snake-DSTM (Saad & Ravindran 2011b) and Hyflow (Saad & Ravindran 2011a) (when in the control flow model) use the D2PC protocol (Raz 1995), an optimized version of 2PC that organizes the nodes in a tree to most efficiently distribute and collect the protocol's messages; the coordinator is determined dynamically as the node to where the messages converge more rapidly with D2PC, thus choosing the optimal coordinator and allowing the earliest possible release of locked resources in each node. Sinfonia (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007) relies on a modified 2PC protocol that piggybacks the minitransaction in the first phase of the protocol. In Dash& Demsky (2011), machines holding the authoritative copies for each object read and written in the transaction check if the version read/written was the latest one and vote accordingly.

Cluster-STM (Bocchino, Adve, & Chamberlain 2008) relies on deadlock detection to abort concurrent transactions as synchronization is performed through (read and write) locks.

DecentSTM's (Bieniusa & Fuhrmann 2010) atomic commitment scheme is based on Paxos (Lamport 1998), a consensus protocol. The various nodes participating in the transaction vote for the transaction to be committed or aborted in the several rounds of the consensus until a decision is made. DecentSTM proposes a randomized distributed consensus protocol. The home node of each object votes on whether a transaction can be committed or not according to the versions read by the objects. A transaction commits if it only receives positive votes; when it receives both positive and negative votes, it asks for another round of consensus. If a home node detects a conflict, it signals all the other nodes with fail messages.

A particular case for managing the data consistency among the nodes is employed by DSMTX (Kim, Raman, Liu, Lee, & August 2010), which relies on a centralized process instead of a distributed protocol, as all systems we have seen so far do. All processes running in the various nodes of the system submit the committing transactions to the centralized process so that they can be validated sequentially. Unfortunately, this approach avoids the need for a complex distributed consistency protocol by creating a bottleneck that potentially hampers the performance of the system.

2.5.3.3 Caching and Prefetching

One of the major challenges of designing distributed shared memory systems is to hide the latency of accessing remote objects. Typical mechanisms used to solve this issue include caching data items on remote nodes in order to try to bypass the overhead of accessing objects stored in other locations, and prefetching which, as the name suggests, consists of pre-obtaining the data items before the transaction needs to access them.

The work presented in Dash & Demsky (2011) implements both caching and prefetching. To ensure that transactions access only the latest version of the objects, the system checks at commit time the versions read. The proposed prefetching mechanism, named symbolic prefetching, circumvents the inefficiencies of traversing remote lists, for instance, by prefetching the entire list from the remote to the local node and minimizing, therefore, the delays due to network latency.

Infinispan (Marchioni 2012) also provides a caching mechanism in which nodes store remote objects locally for a configurable period of time and every time a key is updated, an invalidation message needs to be multicast to ensure nodes with the object in cache invalidate it.

On the other hand, Sinfonia (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007) delegates to the application level the caching mechanisms, taking advantage of the developers' knowledge about the application to create the most appropriate policies.

2.5.4 Replication

Replication is a means to not only provide fault-tolerance to applications, but also achieve scalability. This subsection addresses two aspects of replication, namely replication degree and replica consistency protocols. Different degrees of replication serve different purposes, so one important design decision is the number of replicas a system will have. The choice of the data consistency protocol also plays a major role in the design of a DSTM as it will define how well the system can cope with the different workloads it is subjected to.

2.5.4.1 Replication Degree

A system can be fully or partially replicated to guarantee fault tolerance and scalability.

Full replication increases the overall system dependability, resulting in robust systems but with expensive update transactions. In these systems, every node in the system has an updated copy of every object. Read and write transactions may be processed by all nodes without requiring expensive communication with remote nodes during transaction execution. In the end of each write transaction, however, nodes must coordinate to ensure that all copies of the objects are updated. Such systems include (Couceiro, Romano, Carvalho, & Rodrigues 2009; Carvalho, Romano, & Rodrigues 2010; Palmieri, Quaglia, & Romano 2010).

On the other hand, partial replication can be used as an additional mechanism for fault-tolerance but still allowing highly scalable systems. In these systems, objects are replicated in a pre-determined number of nodes (the replication degree) which, in the end of each write transaction, must coordinate to ensure all replicas are updated, as seen in (Ruivo, Couceiro, Romano, & Rodrigues 2011; Zhang & Ravindran 2011; Schütt, Schintke, & Reinefeld 2008; Kotselidis, Lujan, Ansari, Malakasis, Kahn, Kirkham, & Watson 2010; Marchioni 2012; Zhang & Ravindran 2011).

There are also systems that are fully replicated but only for scalability reasons, such as DMV (Manassiev, Mihailescu, & Amza 2006) and DiSTM (Kotselidis, Ansari, Jarvis, Lujan, Kirkham,

& Watson 2008). DiSTM and one of DMV's consistency protocols rely on a master node to keep the cached data up-to-date. If this node fails, the most recent snapshot of the system is lost and the other nodes may be storing stale data, possibly even in different states that are inconsistent among themselves because some nodes may have received the updates while others did not. The second approach to maintain consistency in DMV does not have a single point of failure as all nodes can execute update transactions, but since the propagation of updates is not based on a reliable communication primitive, nodes may end up with inconsistent data snapshots and even apply updates for transactions that should have been aborted. The replica consistency protocols of these systems will be detailed in the next subsection.

2.5.4.2 Consistency Protocols

In replicated systems, consistency protocols must ensure that all nodes participating in the transaction apply the same updates on the replicated objects in the same or equivalent order. The next paragraphs detail how this is processed in the systems introduced in the previous paragraphs.

D^{2}STM (Couceiro, Romano, Carvalho, & Rodrigues 2009) proposes a new certification scheme, the Bloom Filter Certification (BFC), which consists in encoding the read set of a transaction in a Bloom Filter (Bloom 1970), a space-efficient data structure that allows strongly (and efficiently) compressing the messages disseminated via the TOB service, while still allowing every replica in the system to deterministically certify the transactions. On the down side, BFC can suffer from false positives due the probabilistic nature of Bloom filter-based encoding, which ultimately lead to an additional rate of aborted transactions.

As for partially replicated systems, (Ruivo, Couceiro, Romano, & Rodrigues 2011) proposes a total order multicast primitive. This primitive is used to guarantee the serialization order of the transactions by involving only the nodes that store a replica of the object. Unlike D^{2}STM, this approach implements a weaker consistency model and, therefore, can avoid disseminating read sets for validation. However, as nodes do not store all the data, some transactions cannot be executed locally and the missing data must be fetched from remote nodes, incurring in additional communication costs.

Transactions in Anaconda (Kotselidis, Lujan, Ansari, Malakasis, Kahn, Kirkham, & Watson 2010), another partially replicated system, acquire read and write locks for the objects read and written. The write set is sent to the remote replicas, which abort local transactions in case of conflicts if the latter are younger than the first; deadlocks are also solved by checking which transaction has the highest priority, i.e., which started first.

The work in (Carvalho, Romano, & Rodrigues 2010) proposes ALC, a new certification protocol based on the notion of asynchronous lease. In order to commit a transaction T, a node must own a"lease" on the data items accessed by T. If a node, at the moment of committing T, does not already own a lease on T's read and write sets, it requests the lease via TOB. Once the lease has been acquired, transactions are guaranteed to be sheltered from remote conflicts. This approach has two main benefits. Firstly, it reduces the abort rate of long running transactions. Secondly, if a transaction T executes on a node that already owns a lease for the data items accessed by T, it can disseminate its updates using Uniform Reliable Broadcast, which is a less expensive communication primitive than TOB. This work was later extended by the Lilac-TM system (Hendler, Naiman, Peluso, Quaglia, Romano, & Suissa 2013), which uses adaptive policies to determine whether to issue a lease request or migrate the transaction to the transaction owner.

AGGRO (Palmieri, Quaglia, & Romano 2010) introduces an optimization to the active replication scheme presented in Section 2.1, by maximizing the overlap between communication and transaction processing as it optimistically processes transactions without waiting for the completion of the TOB-based replication coordination scheme. Transactions are processed as they arrive in the Optimistic TOB channel, which corresponds to the spontaneous order by which messages are delivered by the network (and, in LANs, this will most likely correspond to the final total order (Kemme, Pedone, Alonso, Schiper, & Wiesmann 2003)). A related approach is the one adopted in SCert (Carvalho, Romano, & Rodrigues 2011b): also in this case transactions are processed speculatively in order to amortize the latency of an TOB-based coordination scheme. However, unlike in AGGRO, SCert does not target an actively replicated transactional system, but introduces speculative transaction processing techniques over a base system in which transactions are processed according to a certification-based approach.

DMV (Manassiev, Mihailescu, & Amza 2006) implements two consistency protocols which differ in the number of nodes that can execute update transactions. The first one, update-anywhere, is similar to the certification techniques. At commit time, each write transaction creates a new version of the transactional memory, broadcasts the modifications made by itself to all the other nodes and, finally, waits for their acknowledgements before committing the transaction locally. To enforce a consistent serialization order of update transactions, each update transaction obtains a unique system-wide token during commit so that only one node can perform a modification broadcast at any given time. A conflict is detected when a local write transaction is executing and an update from another node arrives modifying a page read or written by the local transaction. In this case, the local transaction is aborted and restarted. The second consistency protocol relies on a scheduler aware of the type of transactions to distribute them among the nodes and is based on the primary-backup replication technique described in Section 2.1. Write transactions can only be executed in a master node and read-only transactions are executed in the slaves. The master node broadcasts the modifications resulting from write transactions to the set of slaves. The global serialization order is decided by the master's internal concurrency control.

DiSTM (Kotselidis, Ansari, Jarvis, Lujan, Kirkham, & Watson 2008) also provides multiple consistency protocols. The first is a decentralized implementation of Transactional Coherence and Consistency (TCC) (Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, & Olukotun 2004). Nodes acquire a global serialization number from the master to serialize the transactions and send the read and write sets to the other nodes for validation. After the commit, the node updates the global dataset kept at the master node which, in turn, eagerly updates all the cached datasets of the rest of the nodes of the cluster. The other two protocols are based on leases. The first one is also a master-centric approach, in which nodes request the lease from the master to serialize the order by which it will be committed, in order to avoid sending the read and write sets. In the second leases protocol, there are multiple leases that can be acquired by several transactions simultaneously if they do not conflict with each other.

Finally, similarly to the previously described DecentSTM, Scalaris (Schütt, Schintke, & Reinefeld 2008; Schintke, Reinefeld, Haridi, & Schütt 2010) implements an adapted version of the Fast Consensus Commit protocol (Gray & Lamport 2006), in which a group of replicated transaction managers collect the votes issued by the nodes storing the replicas of the data times accessed during the transaction; as a group of replicas is used instead of a single transaction manager, this protocol allows to commit transactions when the network becomes partitioned due to some network failure, as long as there is a primary partition (i.e., a partition with the

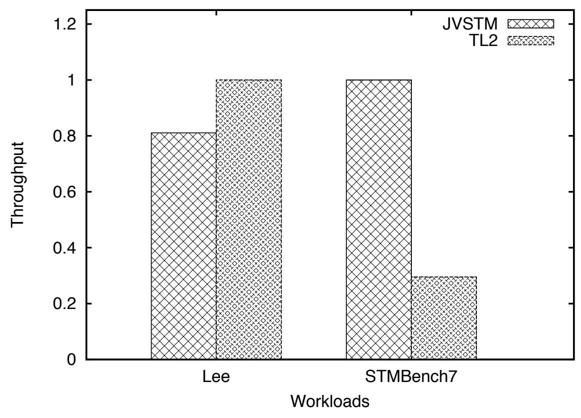


Figure 2.1: Throughput of different STM systems.

majority of nodes).

2.5.5 Discussion

Table 2.2 presents a summary of the systems described in this document. It is very clear that, apart from the nature of transactions supported by the systems and the data granularity (where dynamic transactions and object level granularity predominate, respectively), systems resort to implementing techniques that are quite distinct.

However, there is no perfect combination of techniques that suits all purposes and situations because one will always have to face tradeoffs when designing a system. Let's first consider the example of data distribution versus replication: on one hand, distribution scales better as there is no need for any expensive replica consistency schemes; on the other hand, it sacrifices fault-tolerance for performance since, if one node fails, all the data it was responsible for is lost. Other design choices whose performance can be strongly affected by the actual application workload include the local concurrency control mechanism (Carvalho 2011) as well as the protocol employed for coordinating replicas (Couceiro, Romano, & Rodrigues 2011), which will be described in more detail over the next paragraphs.

The work by Carvalho (2011) studies the influence of the local contention strategy on the performance of the system. It tests two different STMs: JVSTM (Cachopo & Rito-Silva 2006) and TL2 (Dice, Shalev, & Shavit 2006). The only common point between the two systems is their reliance on a commit time write back phase. JVSTM is a multi-versioning scheme, which takes an optimistic approach that detects conflicts only at commit time via a read-set

	Data vs. Control Flow	Static vs. Dynamic Txs	Data Granularity	Data Placement	Caching and Prefetching	Replication Degree	Consistency Protocol
Aggro [Palmieri et al., 2010]	(Full Replication)	Dynamic	Object	(Full Replication)	(Full Replication)	Full	Optimistic Active Replication
ALC [Carvalho et al. 2010]	(Full Replication)	Dynamic	Object	(Full Replication)	(Full Replication)	Full	Asynchronous lease
Anaconda [Kotselidis et al., 2010]	Control	Dynamic	Object	Static	Caching	Partial	Certification-based
Ballistic [Herlihy and Sun, 2007]	Data	Dynamic	Object	Dynamic	Caching	(Not supported)	Local Contention Manager
Cluster-STM [Bocchino et al., 2008]	Hybrid	Dynamic	Word	Dynamic	Caching (application)	(Not supported)	Lock-based
Combine [Attiya et al., 2010]	Data	Dynamic	Object	Dynamic	Cache	(Not supported)	Local Contention Manager
D2STM [Couceiro et al., 2009]	(Full Replication)	Dynamic	Object	(Full Replication)	(Full Replication)	Full	AB-based Certification
[Dash and Demsky, 2011]	Control	Dynamic	Object	Static	Both	(Not supported)	2PC
DecentSTM [Bieniusa and Fuhrmann, 2010]	Control	Dynamic	Object	DHT	(Not supported)	(Not supported)	Paxos-based
DiSTM [Kotselidis et al., 2008]	(Full Replication)	Dynamic	Object	(Full Replication)	(Full Replication)	Full	TCC and leases
DMV [Manassiev et al., 2006]	(Full Replication)	Dynamic	Page	(Full Replication)	(Full Replication)	Full	Certification and Primary-backup
DSMTX [Kim et al., 2010]	Control	Dynamic	Page/Word	Static	(Not supported)	(Not supported)	Centralized component
Hyflow [Saad and Ravindran, 2011a]	Hybrid	Dynamic	Object	Dynamic	Caching (Data flow)	(Not supported)	D2PC (control flow)
Infinispan [inf, 2012]	Control	Dynamic	Object	Consistent Hashing	Caching (optional)	Partial	2PC
Quorum Rep. [Zhang and Ravindran, 2011]	Data	Dynamic	Object	Dynamic	Caching	Partial	Quorum-based protocol
Relay [Zhang and Ravindran, 2009]	Data	Dynamic	Object	Dynamic	Caching	(Not supported)	Local Contention Manager
Scalaris [Schutt et al., 2008]	Control	Dynamic	Object	Symmetric Data Replication	(Not supported)	Partial	Paxos-based
Sinfonia [Aguilera et al., 2007]	Control	Static	Word	Application Specific	Application specific	Partial (optional)	2PC-based
Snake-DSTM [Saad and Ravindran, 2011b]	Control	Dynamic	Object	Static	(Not supported)	(Not supported)	D2PC
TOM [Ruivo et al., 2011]	Control	Dynamic	Object	Consistent Hashing	Caching (optional)	Partial	Atomic Multicast based Certification

Table 2.2: Summary of the DSTM systems presented.

validation phase. On the other hand, TL2 is based on a single-versioned scheme and on an efficient encounter time validation strategy that aims at detecting conflicts during transaction execution (i.e., transactions are validated upon each read/write operation). Figure 2.1 (from (Carvalho 2011)) shows the throughput of both approaches (normalized to the largest) when employed with two different benchmarks. It is clear that the two approaches have very distinct performances and none has the highest throughput in both benchmarks (for a more detailed discussion, please refer to the paper).

As for the replica consistency protocols, (Couceiro, Romano, & Rodrigues 2011) compares three certification protocols for fully replicated systems that differ on the i) the size of the message exchanged by the nodes, ii) the validation process, and iii) the number of communication steps. The Non-Voting protocol (Pedone, Guerraoui, & Schiper 2003), when a transaction finishes its execution and is ready to commit, sends a message to all nodes with the read and write set of the transaction, which may originate potentially large messages; the validation process consists in checking if none of the items in the read set were modified by concurrent transactions; and, as all nodes have all the information needed to complete the validation, only one message is required for a transaction to be certified. The second communication protocol, Bloom Filter certification (Couceiro, Romano, Carvalho, & Rodrigues 2009), previously described in Section 2.5.4.2, also requires only one round of messages and the validation consists in checking if no item in the write set of a concurrent transaction is in the Bloom filter of the transaction. Finally, the Voting protocol (Kemme & Alonso 2000) does not require the write set of transactions to be sent to the other nodes, resulting in shorter messages; but, since only one node possesses all the information, two rounds of messages are necessary: the first to disseminate the write set and the second to inform the nodes whether they should commit or abort the transaction, based on the validation performed at the node storing its read set. Figure 2.2 (from (Couceiro, Romano, & Rodrigues 2011)) depicts the throughput of the three replication protocols. The transactions in the benchmark used do not conflict with each other, each writes two data items and read an increasing number of items: 10, 100 and 10.000. The results clearly show that one protocol is more adequate than the others for each workload (for a more detailed discussion, please refer to the paper).

These facts clearly point to solutions combining more than one approach in several dimensions of a DSTM. GenRSTM (Carvalho, Romano, & Rodrigues 2011a) paves the way for a

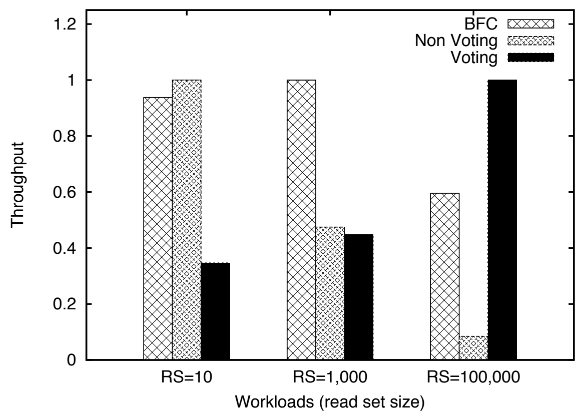


Figure 2.2: Throughput of different replica consistency protocols.

generic, fully replicated, DSTM able to change both the local contention strategy (in this case, encapsulated in the local STM) and the replica consistency protocol.

2.6 Adaptation in Software Transactional Memory

As hinted in the previous section, there is no "one size fits all" STM algorithm or configuration for all workloads. One possible solution is to hard-code all the possible configurations that maximize the throughput of the system for every possible workload, which will require a very exhaustive specification of all the possible combinations that will most likely be incomplete and result in poor performance. Therefore, it is preferable to build systems that are able to self-adapt in order to automatically find the most appropriate configuration for the current workload or environment.

This section presents some of the most representative works in adaptive STMs that address this problem. For both adaptive STM and TM systems, first a meta-architecture is described, followed by a description of each system according to the architecture and ending with a comparative analysis of the systems.

2.6.1 Meta-Architecture of an Adaptive STM

Figure 2.3 depicts a meta-architecture for STM systems. The component that encapsulates the STM algorithm has many different implementations, each more appropriate for a given pattern of read and write accesses. Each implementation has multiple internal parameters that

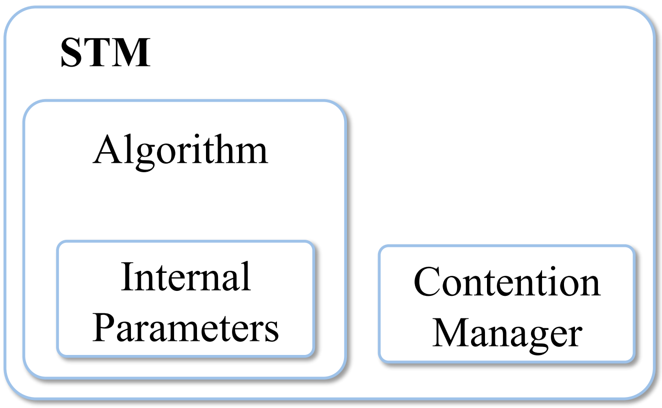


Figure 2.3: Meta-architecture for an adaptive STM system.

can be configured, resulting in different performances according to the workload. Finally, the contention manager enforces priority among transactional accesses, deciding whether aborting a conflicting transaction will improve the overall performance; it then determines the amount of time the transaction must wait before re-trying to execute and commit, thus controlling the contention level of the data.

The next subsection presents systems that perform adaptation within an STM. In Section 2.6.2.1, systems that self-configure internal parameters of the STM are described. The following subsection presents schemes to change the STM algorithm itself. Strategies that take into consideration the optimal concurrency level of the system by varying the number of simultaneous transactions to avoid unnecessary aborts are introduced in Section 2.6.2.3.

2.6.2 Description of Adaptive STM Systems

Section 2.6.2.1 encompasses systems in which the adaptations are within the STM algorithm, either thread-local or system-wide, referred to as fine-grained adaptation. In contrast, systems that implement coarse grained adaptation, described in Section 2.6.2.2, are able to switch the entire STM algorithm, which obviously requires more expensive inter-thread synchronization. Strategies that take into consideration the optimal concurrency level of the system by varying the number of simultaneous transactions to avoid unnecessary aborts are introduced in Section 2.6.2.3. Section 2.6.4 presents two systems that are able to switch between two different implementations of the transaction abstraction.

2.6.2.1 Fine-Grained Adaptation

The configuration of the parameters within an STM algorithm can have a profound impact on the performance of the system: a bad parameterization can dramatically penalize the throughput. The following paragraphs describe three different approaches to tuning the internals of the STM library.

ASTM (Marathe, Scherer, & Scott 2005) The first work on adaptive STM is ASTM, an object-based STM that borrows the best design decisions from the two (at the time) state of the art STM's to build a system that could surpass their performance for all workloads.

ASTM only adapts the acquire semantics (i.e., when to secure the lock to the locations updated by a transaction) as the authors show that the time at which to-be-written locations are acquired has a drastic impact on scalability. The acquire semantics include lazy and eager acquisition. While the first defers the acquisition of locations to the latest possible point in time (i.e., commit-time locking), the latter strives to acquire them as early as possible (i.e., encounter time locking). This work shows that eager acquisition has lower single threaded overhead and, by contrast, lazy acquisition provides better throughput for some multithreaded workloads.

In order to determine which object acquisition strategy to use, the system relies on a history based heuristic based on observations made by past transactions. Rolling averages are maintained for percentages of writes and early releases that follow at least one write across transactions executed by a thread. As long as the first metric is below a given threshold and the second metric above another threshold, transactions use lazy acquire; otherwise they use eager acquire.

TinySTM (Felber, Fetzer, & Riegel 2008) This work is a word-based STM which protects the accesses to shared memory locations with locks. A hierarchical locking scheme is used, on one hand, to speedup the validation stage for transaction with large read sets by allowing each lock to cover a large number of memory locations, and on the other hand, to reduce the probability of false sharing by partitioning locks so that the validation can apply to only the portion of the memory locations actually read by a transaction.

This system allows the self-tuning of several parameters and the three most important are:

1. The hash function to map a memory location to a lock: this parameter allows exploiting the spatial locality of the data structures used by the application as it controls the number of contiguous addresses mapped to the same lock;

2. The number of entries in the lock array: a small value maps more addresses to the same lock and decreases the size of the read sets, but increases the abort rate because of false sharing;

3. The size of the array used for hierarchical locking: a high value increases the number of atomic operations but reduces the validation overhead and contention on the array's elements.

A hill-climbing algorithm with memory and forbidden areas is used to self-adjust the above triple; this algorithm stores the previously tested combinations and the resulting throughput, avoiding combinations of parameters that fail to perform above a given threshold. The strategy makes a move (which can be increasing/decreasing one or more parameters) and then verifies its effectiveness. Moves can be reversed or even forbidden if the drop in the throughput is too high, according to some predetermined thresholds.

AdaptSTM (Payer & Gross 2011) Similarly to TinySTM, AdaptSTM allows the self-tuning of parameters within the STM algorithm. However, while on the former system these parameters are adjusted system-wide, in adaptSTM each thread has its own configuration; the disadvantage of this approach is that some parameters, such as the number of entries in the lock array, cannot be changed in runtime without global synchronization as they are shared by all threads, which is also the main disadvantage of TinySTM.

AdaptSTM allows the following parameters to be tuned:

1. Write-buffering: the abort rate is sampled in order to decide which is the best strategy. In a contended environment with high abort rate, it is more beneficial to use write-back instead of write-through to commit the changes to memory as it is cheaper to abort transactions in the former.

2. Size of the hash table: this parameter determines the number of memory locations assigned to an entry in the hash table containing the locks (read set and write set). If the table is

too small, the lookup will be slow due to the hash collisions; else, the overhead for resetting the table every time a transaction starts is high.

3. The hash function that maps a memory location to a lock: as discussed in TinySTM, this parameter trades off quality (i.e., higher or lower probability of false sharing) for speed.
4. Contention management: the adaptive contention manager implements a backoff strategy which retries immediately if the contention is low and yields in an increasing amount of time otherwise.

The tuning of these parameters is performed according to predetermined thresholds and moving averages for several metrics collected from the running system. The metrics include the transaction frequency, number of unique read and write locations, number of hash table collisions per hash table, abort rate and the quality of the hash functions.

2.6.2.2 Coarse-Grained Adaptation

In some occasions, tuning only parameters within STM algorithms does not provide enough variety to cope with all the possible changing workloads. So, coarse-grained adaptation enables a system to choose from a library of algorithms the one that fits the workload the best.

Spear (2010) This work allows both fine and coarse-grained adaptivity. At the fine-grain level, they implement read-only optimizations in which transactions first start as optimized read only transactions and, when the first write occurs, the mode of the transaction is changed. As for the coarse-grained adaptativity, the system is able to switch among different STM algorithms.

The adaptation is performed according to a set of predetermined policies. A state machine is employed that takes as input several system metrics as well as constraints the programmers may indicate regarding transaction semantics, for instance, whether the transaction can be retried or not.

Wang et al. (2012) While the previous system reacted to bad performance by making decisions based on the likelihood of pathology and precision of conflict detection, this work (Wang, Kulkarni, Cavazos, & Spear 2012) develops a system that can be trained in the environment in which it will be deployed so that the STM can automatically tune its performance based on the operating environment and workload characteristics instead of the fixed policies of the previously studied systems.

Similarly to Spear (2010), the adaptation is performed by selecting the most appropriate STM algorithm from a library of different algorithms.

The main difference between the approaches of Wang et al. (2012) and Spear (2010) is how the choice for the most appropriate algorithm is made. While Spear (2010) relies solely on expert policies, the work by Wang et al. (2012) relies on machine learning to trigger adaptativity. The algorithm collects metrics in runtime and can be trained either with a set of micro-benchmarks or with the application itself. In addition, expert policies can be employed; the machine learning output is able to enrich them by providing the correct configuration in the cases the policies fail.

2.6.2.3 Contention Manager Adaptation

One other dimension to be considered when adapting an STM is the contention manager. When the workload lacks inherent paralellism, launching an excessive number of transactions can adversely degrade performance. This subsection presents four different approaches to adaptivity in the context of contention management.

Yoo et al. (2008) Typically, a contention manager can decide to abort a certain transaction but it does not deal with when to resume the aborted transaction (Scherer III & Scott 2004). In (Yoo & Lee 2008), incoming transactions are put in a queue when an indicator, the contention intensity, exceeds a pre-established threshold. The queue dispatches one transaction at the time. If the contention intensity is below the threshold, transactions skip the queue; otherwise, they have to wait until a scheduler signals them to proceed. In an extreme case, when all transactions are queued, this mechanism turns transactional behaviour into a single coarse-grained lock. The contention intensity is dynamically calculated from the number of aborted and committed transactions. More recently, this approach was extended by Tuner (Diegues & Romano 2014), which relies on reinforcement learning techniques to determine when transactions should be serialized: this relieves programmers from having to manually determine the adequate setting of the threshold on the contention intensity.

Ansari et al. (2011) While in the previous work running transactions were put on hold in case of high contention, the authors of (Ansari, Luján, Kotselidis, Jarvis, Kirkham, & Watson 2011) rely on control theory to build an algorithm that dynamically changes the number of active threads which can concurrently execute transactions considering the observed transaction conflict rate. The transaction conflict rate is the percentage of committed transactions out of all executed transactions in a sample period. The number of active threads is decreased when this rate exceeds a threshold while it is incremented when it lower than another threshold.

Rughetti et al. (2012) This solution (Rughetti, Di Sanzo, Ciciani, & Quaglia 2012) relies on machine learning to determine the optimal number of threads executing concurrent transactions as a function of the current data access pattern. This is calculated by estimating the average read and write set sizes, execution time for committed transactions, execution time of non-transactional code and the probability of conflict upon executing a read or write operation. This is more flexible that the previous two approaches, as it does not rely on pre-determined thresholds but on Artificial Neural Networks (Haykin 1994). A similar approach was taken also by Didona et al. (Didona, Felber, Harmanci, Romano, & Schenker 2013), which, however, relied on a model-free black-box approach based on hill-climbing.

Maldonado et al. (2011) Some applications require bounded response time for some of their operations. The work by (Maldonado, Marlier, Felber, Lawall, Muller, & Riviere 2011) tackles this problem by adapting the contention mode of a transaction according to the time left for its deadline. As the deadline approaches, the sequence of execution modes is the following: optimistic mode (i.e., write locks are acquired upon commit and the updates are written-back upon successful validation), visible reads (i.e., an update transaction may abort if it detects a conflict with a read-only transaction, favouring the read-only transaction executing in this mode versus an update transaction in the optimistic mode), and irrevocable (i.e., the transaction is protected from aborts and is guaranteed commit). Transactions can execute concurrently in these three modes.

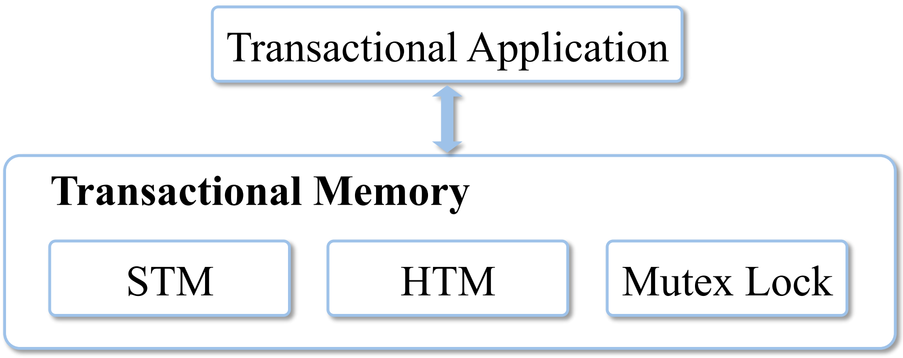


Figure 2.4: Meta-architecture for an adaptive TM system.

2.6.3 Meta-Architecture of an Adaptive TM

On a higher level, a transactional memory system can have several implementations of the transaction abstraction, each favouring specific workloads. Figure 2.4 depicts the meta-architecture of an adaptive TM system which is able to switch between software and hardware transactional memory and mutex locks. The next subsection presents two systems that change the TM implementation according to the current workload characteristics.

2.6.4 Description of Adaptive TM Systems

Usui et al. (2010) In this work, the authors propose a system that dynamically decides whether a critical section can be executed as a (software) transaction or while holding a mutex lock. The rationale behind this adaptation is that using a coarse-grained lock is more efficient when the degree of parallelism is low but, on the other hand, transactional memory allows for a better throughput when there are more concurrent transactions. The decision for when to switch is very simple: the system estimates the overhead for running in transactional mode and the switch is made if the value is below a given threshold. This cost-benefit analysis takes into account the number of threads contending for the atomic block, the number of times a transaction must try before it commits, and the transactional overhead.

PhTM (Lev, Moir, & Nussbaum 2007) This system supports different transactional memory implementations by separating the execution of the application in phases. The authors implement an hybrid transactional memory system that is able to switch between software and hardware transactional memory (HTM). While HTM incurs in lower execution time overhead, it suffers from strict resource limits; STM, on the other hand, does not have such limitations but the overhead for manipulating transactional objects and maintaining multiple versions of transactional data is significant. PhTM preferably executes transactions in HTM, but if the collected metrics show that HTM is no longer the most appropriate implementation based on pre-determined thresholds, the system switches to a phase in which transactions are executed in software.

2.6.5 Comparison of the Approaches

This subsection considers three relevant aspects when building an adaptive STM: how the mechanism is triggered, how the next configuration is determined, and how the self-configuration is implemented. Each of the previously described systems will be classified according to the way these aspects are implemented.

2.6.5.1 Triggering the Adaptation

In the studied systems, the most popular mechanism to trigger adaptation is to use static, hard-coded policies, typically based on a pre-defined threshold. Additionally, machine learning tools, the distance to a deadline and cost-benefit analysis are also employed.

Most systems (Marathe, Scherer, & Scott 2005; Payer & Gross 2011; Felber, Fetzer, & Riegel 2008; Spear 2010; Yoo & Lee 2008; Ansari, Luján, Kotselidis, Jarvis, Kirkham, & Watson 2011; Lev, Moir, & Nussbaum 2007) rely on pre-defined thresholds (usually estimated from the analysis of previous runs) to detect if the current configuration no longer achieves the highest possible throughput for the application. This technique generally takes into consideration workload metrics (number of reads/writes, abort rate, etc.) and system characteristics (number of cores, number of threads executing, etc.).

Works by (Wang, Kulkarni, Cavazos, & Spear 2012) and (Rughetti, Di Sanzo, Ciciani, & Quaglia 2012) rely on machine learning algorithms. These algorithms are trained with generic benchmarks or with the application itself and, given the current workload conditions, are able to predict the best performing configuration and trigger adaptation in case it does not match the current one.

The work by Maldonado et al. (2011) deals with transactions that must commit before a given deadline, which means that the trigger for adaptation is how close the transaction commit is to its deadline.

Finally, Usui et al. (2010) perform a cost-benefit analysis for each critical section that takes into consideration the number of threads contending for that critical section, the number of times a transaction needs to try before it commits and the transactional overhead.

2.6.5.2 Determining the Optimal Configuration

A significant number of the studied systems (Marathe, Scherer, & Scott 2005; Payer & Gross 2011; Yoo & Lee 2008; Lev, Moir, & Nussbaum 2007) rely on previously determined and hard-coded thresholds to determine the best performing configuration. The following paragraphs describe the remaining systems' distinct approaches to tackle this issue.

TinySTM (Felber, Fetzer, & Riegel 2008) uses a hill-climbing algorithm to determine the new values for the parameters, together with the thresholds. The throughput is measured over one second and associated with each tuning configuration. Then, the hill-climbing algorithm decides the new configuration, which is evaluated throughout the next period.

The work of Spear (2010) relies on a set of user defined policies to select the most appropriate STM algorithm. The system takes as input several parameters concerning mostly the number and frequency of aborted transactions and also the applications requirements regarding the STM implementation characteristics (whether it supports self-abort of transactions, for instance) to determine the most adequate policy for each phase of the execution.

The authors of (Ansari, Luján, Kotselidis, Jarvis, Kirkham, & Watson 2011) propose a controller model that determines the optimal number of concurrent threads considering the current transaction rate but the programmer must pre-establish how conservative the controller will be towards resource usage efficiency.

Both Wang et al. (2012) and Rughetti et al. (2012) rely on a more flexible way to determine the next best system configuration than fixed thresholds and user-defined policies: machine learning. The first system can solely rely on the algorithm's output to determine the most appropriate configuration or combine it with user-defined policies to act as a fallback when the policies fail. As for the second system, the various outputs of the machine learning tool (each corresponding to the predicted throughput of each possible configuration) are used to determine the optimal number of concurrent threads.

Similarly to how the adaptation is triggered, the deadline-aware system of Maldonado et al. (2011) determines the next transaction configuration according to how close the end of the transaction is to its pre-defined deadline.

The previously mentioned cost-benefit analysis of Usui et al. (2010) determines which configuration is the best for each atomic block, whether it should be executed as a transaction or with the lock-based implementation.

2.6.5.3 Implementing the Adaptation Mechanism

There are two main approaches for implementing the adaptation mechanism. On one extreme, the execution of transactions must halt during the entire configuration switch; on the other, transactions are completely independent, different transactions can be simultaneously executed with distinct configurations. Other systems decide on a per-case basis how the system behaves when the adaptation is underway.

Nearly all the studied systems must stop the execution of transactions when reconfiguring (Felber, Fetzer, & Riegel 2008; Wang, Kulkarni, Cavazos, & Spear 2012; Lev, Moir, & Nussbaum 2007). In (Payer & Gross 2011) and (Spear 2010), the system stops unless the configuration is for a thread local parameter.

In ASTM (Marathe, Scherer, & Scott 2005), when a new object acquisition scheme is decided, the new transactions will use it, without requiring old transactions to be finished in the meantime.

Transactions in (Maldonado, Marlier, Felber, Lawall, Muller, & Riviere 2011) and (Usui, Behrends, Evans, & Smaragdakis 2010) are independent, which means that different execution modes can be active at the same time in the system and, when an adaptation occurs, the system as a whole does not need to stop running new transactions.

As for the systems that self-configure the number of concurrent transactions, the work of Yoo et al. (2008) queues the stopped transactions and signals them when the contention intensity lowers to an acceptable level. The authors of (Ansari, Luján, Kotselidis, Jarvis, Kirkham,

& Watson 2011) and (Rughetti, Di Sanzo, Ciciani, & Quaglia 2012) start and stop threads according to the needs, without interfering with the others running their own transactions.

2.6.5.4 Discussion

Figure 2.3 presents a summary of the systems presented regarding the three aspects discussed above. There are two clear tendencies: the adaptation is triggered by a metric reaching a threshold and the system must stop in order to change the configuration.

Comparing pre-determined thresholds with machine learning techniques that are trained with either the target application's workload or a more generic one, it can be easily concluded that the former strategy is significantly less flexible than the latter as manually determining the settings of thresholds-based systems is typically a very complex and time-consuming task. Machine learning based solutions, conversely, remove this burden from the user/system administrators, by automating the learning of the appropriate parameters' settings.

As for the second main design choice, systems may either stop executing transactions when the system is reconfigured or implement mechanisms that allow for more than one configuration to be running simultaneously. As a result, the system can perform more fine-grained adaptations without incurring in great throughput losses.

In conclusion, from the above analysis, an ideal adaptive system should have a triggering mechanism sufficiently flexible to adjust to workload variations that were not foreseen when training the system. In addition, halting the execution of transactions is not desirable as it hampers the system's throughput, so a solution like (Marathe, Scherer, & Scott 2005), in which different modes of execution co-exist, should be the most appropriate.

2.7 Summary

This chapter presented the main concepts of the four fundamental areas that are on the core of the presented work, and also several approaches to adaptivity in STM's, from the tuning of some parameters in STM algorithms to switching between STM and hardware transactional memory.

	What is adapted	Adaptation trigger	What determines the next configuration	How the self-configuration is implemented
ASTM [Marathe et al., 2005]	Internal parameters	Pre-determined threshold	Parameters adjusted according to the metrics and thresholds	The following transactions will use the new configuration
AdaptSTM [Payer et al., 2011]	Internal parameters	Pre-determined threshold	Parameters adjusted according to the metrics and thresholds	The system stops, except if it is a thread local parameter
TinySTM [Felber et al., 2008]	Internal parameters	Pre-determined threshold	Hill-clilmbing algorithm	The system stops executing new transactions
[Spear, 2010]	Internal parameters and full algorithms	Pre-determined threshold	User-defined policies	The system stops, except if it is a thread-level parameter
[Wang et al., 2012]	Full algorithms	Machine learning tool and/or pre-determined threshold	The machine learning output and/or user-defined policies	The system stops executing new transactions
[Yoo et al., 2008]	The number of concurrent transactions	Pre-determined threshold	Pre-determined threshold	Transactions are put on hold until they can continue executing
Tuner [Diegues et al., 2014]	The number of allowed retries before transactions are sequentialized	Reinforcement learning	Reinforcement learning	Transactions are retried in hardware before being executed in software
[Ansari et al., 2011]	The number of concurrent threads executing transactions	Pre-determined threshold	Controller model	Threads are activated or deactivated when transactions end
[Rughetti et al., 2012]	The number of concurrent threads executing transactions	Machine learning tool	Machine learning output	Threads are activated or deactivated when transactions end
[Didona et al., 2013]	The number of concurrent threads executing transactions	Reinforcement learning/ hill climbing	Machine learning output	Threads are activated or deactivated when transactions end
[Maldonado et al., 2011]	Execution mode of transactions	How close the deadline is	How close the deadline is	Transactions are independent
[Usui et al., 2010]	Lock-based vs STM	Cost-benefit analysis	Cost-benefit analysis	Transactions are independent
PhTM [Lev et al., 2007]	STM vs HTM	Pre-determined threshold	Parameters adjusted according to the metrics and thresholds	The system stops executing new transactions

Table 2.3: Summary of the Adaptive STM systems presented.

From this analysis of existing literature, however, it emerges the lack of self-tuning solutions for replicated STMs. On the other hand, the problem of adaptivity in replicated STMs cannot be tackled simply by leveraging on existing self-tuning techniques developed for non-distributed STMs. Due to their distributed nature, in fact, replicated STMs raise a host of new challenges, such as dealing with the issues associated with network latency, failures and asynchrony. So, the following chapters propose an autonomic replicated STM system which is able to adapt to several workloads through self-configuration, presenting not only its architecture but also the possible autonomic mechanisms that decide when to adapt.

3 Performance Prediction of Total Order Broadcast Protocols

Total Order Broadcast (TOB) (Defago, Schiper, & Urban 2004) is a fundamental building block for developing strongly consistent replicated systems. TOB greatly simplifies the development of fault-tolerant applications by ensuring that messages are delivered at all replicas in the same order despite variable communication delays and the occurrence of failures, hiding the issues associated with enforcing system-wide agreement on the streams of updates generated by the replicas. TOB is, in fact, at the heart of the classic, general-purpose, active replication scheme (Schneider 1993), as well as of a number of specialized replication protocols tailored for, e.g., database systems (Pedone, Guerraoui, & Schiper 2003) and transactional memories (Couceiro, Romano, Carvalho, & Rodrigues 2009).

Over the last decades, a wide body of literature has been devoted to the design and evaluation of TOB protocols (extensively surveyed by Defago et al. (Defago, Schiper, & Urban 2004)). However, we are not aware of any work proposing engineering methods and tools capable of providing real-time, fine-grained (i.e. on a per message basis) forecasts of the performance of TOB protocols, when deployed in real systems and subject to complex workloads.

This chapter presents the challenges associated with using machine learning techniques to derive fine-grained performance prediction models of total order broadcast (TOB) protocols. The machine-learning based approach proposed allows forecasting the TOB's latency on a per message basis, providing a fundamental building block for architecting self-optimizing replication schemes (Romano, Carvalho, & Rodrigues 2008).

The chapter starts by presenting a semi-opaque self-monitoring architecture that relies on the tracing of a basic set of protocol-independent performance metrics, possibly augmented with protocol-specific context information in a modular fashion via the use of standard programmatic interfaces. This generic (i.e. protocol independent) monitoring tools track the usage of system resources (such as network bandwidth, CPU and memory) across multiple time scales spanning several orders of magnitude. This allows to combine information representative of stationary phenomena (captured by long term averages) as well as transient burstiness (captured by short term averages) which can significantly affect the latency of ongoing TOBs.

Next we discuss the results of an extensive experimental study based on:

- three machine learning methods, namely neural networks (Haykin 1994), support vector regression (Shevade, Keerthi, Bhattacharyya, & Murthy 2000), and regression decision trees (Quinlan 1992).
- three highly heterogeneous and demanding (in terms of amount of injected traffic) workloads, consisting of a synthetic traffic generator that allows us to widely span in the workload's parameters space, and two complex applications running on top a distributed software transactional memory platform (Couceiro, Romano, Carvalho, & Rodrigues 2009) that generate high contention on the computational and memory resources locally available at each node.
- two different TOB algorithms relying on radically different approaches for establishing agreement on the delivery order (centralized vs distributed) and aiming at optimizing distinct performance metrics (latency vs throughput).

These experimental results highlight that the set of context information (also called features in the machine learning literature and in the remainder of this chapter) that maximizes the machine learners accuracy varies significantly when one considers heterogeneous, realistic workloads. We also evaluate to what extent incorporating time series, protocol dependant information and garbage collection metrics can allow enhancing the accuracy of the machine learners.

We then focus on the issue of feature selection, a problem of combinatorial nature that becomes rapidly intractable in scenarios characterized by a large abundance, and redundancy of input variables, such as the one evaluated in this chapter. This experimental data highlights that, while being certainly more efficient than an exhaustive exploration of the feature space, existing heuristics approaches (Guyon & Elisseeff 2003) to the feature selection problem still have prohibitively high execution times. This can represent a major impairment in scenarios demanding frequent re-training of the performance predictors, due to, e.g., workload fluctuations or alterations of the group size caused by failures or dynamic expansions/contractions triggered by spikes of the load pressure. To tackle this issue we propose and evaluate two alternative solutions:

1. An optimized search heuristic, whose search trajectory in the features' power set is drastically restricted with respect to classical greedy search heuristics. This is achieved by exploring exclusively the combinations of features which were found to generally maximize the accuracy of the machine learners. Such a specialization allows reducing the feature selection execution time on average by a factor 10x at a negligible cost in terms of accuracy degradation (<2%) across the whole spectrum of considered workloads.

2. A technique based on the combination of a small set of models, each one relying on different (and largely non-overlapping) subsets of features, and whose predictions are combined on the basis of the expected confidence intervals of the individual models to operate in the corresponding region of their features space. When compared with classical greedy heuristics for feature selection, this ensemble technique allows boosting feature selection by two orders of magnitude, at the cost of an average 10% degradation of the prediction accuracy.

The remainder of this chapter is structured as follows. In Section 3.1 we present the architecture of the system, discussing the key implementation issues of the real-time monitoring tools. Section 3.2 overviews the machine learners, the workloads and the Total Order broadcast algorithms used in our evaluation study. The results of the experimental evaluation are discussed in Section 3.3. Section 3.4 presents a short discussion on related systems. Finally, Section 3.5 concludes the chapter.

3.1 System Architecture

The architecture of this system is depicted in Figure 3.1. It considers a distributed application, such as a transactional database replication manager (Pedone, Guerraoui, & Schiper 2003) or a distributed transactional memory (Couceiro, Romano, Carvalho, & Rodrigues 2009), which is supported by a group communication service (GCS) (Miranda, Pinto, & Rodrigues 2001). The system is augmented with a monitoring layer and a latency predictor, which are the key contributions of this part of the work. Before detailing the description of these components, we will first provide a brief overview of the interdependencies among the system components and discuss some key principles underlying their design.

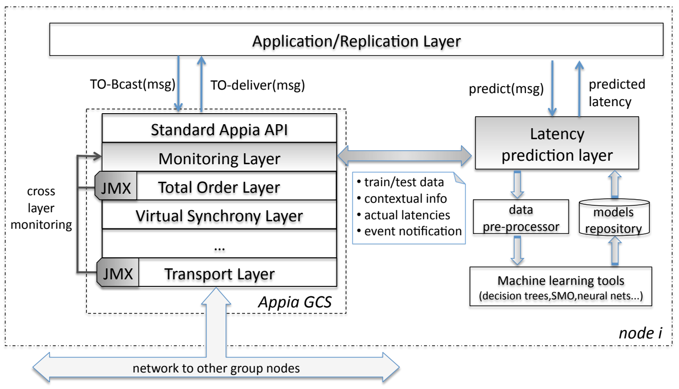


Figure 3.1: Architectural Overview (Single Node Perspective).

In this system, each node develops, in an independent and fully distributed fashion, predictive models of the TOB latency as observed by applications residing on that same node. More specifically, the latency predictor component provides the client of the TOB service with a latency estimator. The predictor is able to forecast the time it takes to self-deliver (after being totally ordered) a message of a given size sent by the application.

The monitoring layer is the component responsible for collecting training data for the machine learners, as well as to provide the latency predictor with information concerning the actual workload characteristics and resource utilization levels. Additionally, it provides feedback to the latency predictor component regarding the accuracy of its forecasts, as well as notifications on the occurrence of relevant changes in the system configuration that may affect the quality of the currently employed predictive model, for instance, a change in the number of active replicas (as the performance of TOB is typically a function of the number of participants).

3.1.1 Monitoring Layer

This monitoring layer has been developed for the Appia (Miranda, Pinto, & Rodrigues 2001) GCS. Appia follows an architectural design that allows to compose layered stacks of micro-protocols according to the application needs. The flow of information among the layers of the Appia stack is supported by the exchange of events that are propagated upwards and downwards through the stack. In Appia, this flow of events is regulated by a single, dedicated thread which we will refer to in the following as Event Scheduler (ES) thread.

The monitoring mechanisms are implemented as a layer that can be transparently disabled/enabled at run-time, ensuring that there is no monitoring overhead when the tracing functionality is disabled. As depicted in Figure 3.1, the monitoring layer sits between the TOB layer and the interface towards the application. This allows to achieve total transparency for the application, as well as to straightforwardly trace any event generated by or delivered to the application. Thus, the monitoring layer is able to intercept TO broadcast/delivery events and events notifying of changes in the group membership. As noted before, membership changes may have a significant impact on the TOB performance, thus they can be used to trigger the generation of a new performance model.

When the monitoring layer is enabled, it collects context information using the following set of metrics:

1. Network related metrics: moving averages across multiple time scales of i) the number of TO broadcast/delivery events, and of ii) the amount of bytes sent/received by the TO layer; additionally it keeps track of the number of TO broadcast events generated by the application layer and for which it has not been generated the corresponding TO delivery event yet.

2. *CPU related metrics:* moving averages across multiple time scales of the total CPU utilization, and of the CPU utilization of Appia's ES thread.

3. *Memory related metrics:* the free memory in the Java Virtual Machine (JVM), as well as two metrics describing the activity of the JVM's Garbage Collector (GC) thread namely, i) the time occurred since the last garbage collection cycle, and ii) the percentage of time elapsed since the last garbage collection cycle with respect to the time between the last two garbage collection cycles. Note that, since there is no standard Java API to directly track the status of the GC thread, to trace the GC activity in a portable manner we extended the *finalize()* method of a dummy object to keep track of the time in which the GC thread is activated (and re-instantiate the dummy object).

In addition to the above context information, the monitoring layer has been designed to support also cross-layer tracing in a modular fashion. Specifically, at system's bootstrap, and upon any alteration of the Appia stack, the monitoring layer queries the whole set of Appia's layers via the standard Java Management Extensions (JMX) interface to determine whether

Metric	**Description**
freeMem	Free memory in the Java Virtual Machine
tLGC	The time since the last garbage collection
pLGC	% of time since the last GC cycle w.r.t. the time between the last 2 GC cycles
undelivMsgs	#TO Broadcast msgs and not yet self-delivered
bytesUp$_x$	#Bytes received over a x msec. time window
bytesDown$_x$	#Bytes sent over a x msec. time window
TOBUp$_x$	#TOB deliver events over a x msec. time window
TOBDown$_x$	#TOB broadcast events over a x msec. time window
totCPU$_x$	% total CPU utilization over a x msec. time window
esCPU$_x$	% CPU utilization by ES thread over a x msec. time window
esCPU$_x$	% CPU utilization by ES thread over a x msec. time window
TCPqueue	Outgoing messages queued at the Transport Layer *(protocol dependant metric traced via JMX interface)*
toTime	Elapsed time since the token was last owned *(protocol dependant metric traced via JMX interface)*

Table 3.1: List of metrics collected by the Monitoring Layer.

there are any layers that externalize information related to their internal state that could be exploited by the machine learners to generate more accurate performance models.

This approach allows developers to specify which attributes, among those monitorable via the JMX interface, should be traced by this monitoring layer as deemed potentially beneficial to enhance the machine learners' accuracy. As we will further discuss in Section 3.2, in the presented experimental analysis we exploit this mechanism to monitor the number of outgoing message queued at the lower layer of the Appia's stack (namely the Transport Layer), as well as to track the state of internal variables of TOB algorithms. We report the whole set of metrics gathered by the monitoring layer in Table 3.1.

Whenever a TO broadcast event for message m is intercepted by the monitoring layer, the latter takes a snapshot of the current state of the context information. As soon as the monitoring layer intercepts the TO delivery event for message m, it determines the self-delivery latency, logs the associated context information (namely the context information at m's sending time along with its self-delivery latency) asynchronously to a memory buffered file and propagates the TO delivery event upwards. The choice of measuring exclusively the self-delivery latencies allows to circumvent the issue of ensuring accurate clock synchronization among the communicating nodes, which would have clearly been a crucial requirement in case we had opted for monitoring the delivery latencies of messages generated by different nodes. Preliminary experiments conducted in our cluster have indeed highlighted that the accuracy achievable using conventional clock synchronization schemes, such as NTP, is often inadequate for collecting meaningful measurements of the TO broadcast inter-nodes delivery latency, being the latter frequently around or less than a millisecond.

3.1.2 Latency Predictor

The latency predictor is responsible for forecasting the performance of the TOB layer when broadcasting a message of a given size. In order to build the performance models used to generate these forecasts, the latency predictor layer triggers the pre-processing of the training data collected by the monitoring layer. In this phase, the training data is prepared (properly filtered and manipulated, e.g., to generate time-series - see Section 3.3.1) to allow its successful processing by the chosen machine learning tool. This system's architecture in fact supports the modular integration of alternative machine learning libraries.

The models output by the machine learners are stored in a model repository, where they are associated with metadata that captures the context in which these models were built (such as the number of machines participating in the TOB group and the TOB algorithm employed while generating the training data). Furthermore, the models are ranked (e.g. for feature selection purposes) and made available to the latency predictor layer for generating performance predictions. Note that the performance models output by the machine learners take as input features not only the size of the message to be broadcast, but also a (possibly quite large) set of system metrics. These are obtained by querying at run-time the monitoring layer. The latter makes also available information on the actual self-delivery latencies of recently broadcast messages, which can be used by the latency predictor to assess the accuracy of its predictions and possibly trigger the construction of a new model.

3.2 Testbed Description

3.2.1 Overview of the Evaluated Machine Learning Methods

We integrated in this system two machine learning tools, namely Rulequest's Cubist (Quinlan b) and Weka (Frank, Hall, Holmes, Kirkby, Pfahringer, & Witten 2005), enabling the user to choose which one to use. We now briefly overview these two tools.

Cubist is a decision tree based regression commercial tool developed by Quinlan, the author of C4.5 (Quinlan 1993) and ID3, two popular decision tree based classifiers. Analogously to these algorithms, Cubist builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain (namely the difference in entropy). Unlike C4.5 and ID3, which contain an element in a finite discrete domain (i.e. the predicted class) as leafs of the decision tree, Cubist places a multivariate linear model at each leaf. An appealing characteristic of Cubist is that the decision tree can be reformulated as set of human-readable rules, where each rule identifies a region in the feature space. Also, each rule contains a multivariate linear model in the "then" clause and is associated with the expected average error in the prediction. Since Cubist generates a piecewise regression model (each multivariate linear model being applicable under certain rules), it can be more powerful than a simple multivariate linear model as it allows variables to be weighted differently as conditions change. When two rules overlap, the values predicted by using the models associated with each rule are averaged with a weight that depends on the degree of confidence in the prediction generated by the two rules.

Weka is an open-source framework providing a common interface to a large number of machine learning algorithms. In this work we evaluate two major regression techniques, namely, Neural Networks (Haykin 1994) and Support Vector Machines (Shevade, Keerthi, Bhattacharyya, & Murthy 2000). These methods are well-known and have been extensively described in the machine learning literature, so we will only briefly overview them. The neural network algorithm implemented in the Weka framework trains a multi-layered network using the classic back-propagation algorithm (Rumelhart, Durbin, Golden, & Chauvin 1995) to determine the weights that minimize the local error at each perceptron. We used the default configuration in Weka, which generates a number of hidden layers equals to half the number of input features. Concerning the Support Vector Machine technique, we also rely on the default configuration of the Weka's SMOreg package, which uses a polynomial kernel whose parameters are learnt using the algorithm in (Shevade, Keerthi, Bhattacharyya, & Murthy 2000).

3.2.2 Workload Description

For this experimental study we consider the following three workloads:

Synth: this is a synthetic benchmark which injects traffic at each node following a regular and homogeneous pattern. On each node we run a single application-level thread which TO broadcasts, during intervals lasting 30 seconds each, messages of growing size, namely {100, 200, 500, 1K, 2K, 5K, 10K, 20K, 50K, 100K, 200K, 500K} bytes, at an increasing sending rate, namely {1, 2, 10, 20, 50, 100, 125, 166, 333, 500, 1000} messages per second. The training and testing datasets are built collecting for each configuration at most 90 messages.

RBTree: this workload is generated by D^2STM (Couceiro, Romano, Carvalho, & Rodrigues 2009), a distributed software transactional memory platform, running the Red Black Tree benchmark. D^2STM uses a TOB-based distributed certification scheme that relies on TOB to propagate the read and write sets of local transactions, and ensure that all replicas validate transactions in the same common order. The Red Black Tree (Herlihy, Luchangco, & Moir 2006) is a well-know benchmark for the evaluation of software transactional memories, in which a red black tree data structure is concurrently updated (by inserting and/or removing items) by several threads. The benchmark was ported to run on the D^2STM platform and tuned to generate transactions entailing a variable number of operations. Note that this has the effect of further increasing the heterogeneity of the generated workload: as the number of operations issued by each transaction varies over time, the frequency of generation of TO broadcasts, and the size of the TO broadcast messages (which encode the transactions' read and write set) also vary accordingly. Unlike the Synth benchmark, in RBTree each replica can host a variable number of threads performing computational intensive tasks before issuing a TO broadcast. This scenario is therefore characterized by a much higher contention among (GCS and application) threads on the local resources (CPU in primis). We will see that this has a significant impact on the predictability of the GCS performance. The data set used to train and test the machine learner consists of the messages exchanged during 15 minutes, which corresponds to the time needed to collect training data across the whole range of the generated workload.

STMBench7: this workload is also generated by D^2STM, running STMBench7 (Guerraoui, Kapalka, & Vitek 2007), a complex benchmark which manipulates an object-graph with millions of objects, featuring a number of operations with different levels of complexity. This benchmark includes both very short and very long-running transactions; the latter traverse hundreds of thousands of objects and generate extremely large read and write sets. As a consequence, the workload for the TO service entails both very short (on the order of few hundreds of bytes) and very large messages (on the order of several megabytes). Also, as transactions need to store in memory their read and write sets, long-running transactions end up stressing significantly the memory system of the local JVM, triggering frequent garbage collection cycles. Like in the RBTree benchmark, this benchmark allows running multiple concurrent application level threads in each node. Each run of this benchmark lasts around 30 minutes in order to ensure that, independently of the number of machines and threads, the training and testing data sets contain approximately 12.000 entries.

3.2.3 Evaluated TOB Algorithms

In these experiments, we consider two classic, well-known TOB algorithms, described, e.g., by Defago et al. in (Defago, Schiper, & Urban 2004). The first one is a sequencer-based algorithm in which a single node, called the sequencer, determines the order according to which all nodes have to TO deliver messages. The second one is a token-based algorithm which ensures agreement on the TO delivery order by circulating among the nodes a token that grants the right to broadcast messages.

The choice of these two algorithms was aimed at maximizing diversity, with the ultimate purpose of widening the representativeness of our testbed. The above TO algorithms, in fact, rely on extremely different approaches for establishing agreement on the delivery order (centralized vs distributed), aim at optimizing different performance metrics (latency vs throughput), and have complementary pros and cons (Defago, Schiper, & Urban 2004).

3.3 Analysis of the Results

In this section we present the results of this experimental study. We will initially focus on the analysis of the results obtained using Cubist and only subsequently move to compare the performance of the Neural Network and SMO methods. All the results reported in the following were obtained using a testbed of nodes equipped with an Intel QuadCore Q6600 at 2.40GHz with 8 GB of RAM running Linux 2.6.27.7 and interconnected via a private Gigabit Ethernet.

The accuracy of the machine learners is measured using the following metrics. Relative Average Error (RAE), which compares the performance of the predictor with that of a naive

Worst vs Best Set of Feat.	**Benchmark**	**NAE**
6 vs 50 time window	Synth (m4)	25%
500 vs 10 time window	RBTree (m2t3)	32%
without vs with GC	Synth (m4)	29%
without vs with time series	RBTree (m2t3)	81%
without vs with token	RBTree (m2t1)	61%

Table 3.2: Models' Accuracy Increase (NAE) with Selected Features.

predictor that simply outputs the average value of the training data. When comparing two different models, say M_1 and M_2, we will rely on the Normalized Additional Mean Absolute Error (NAE), defined as $NAE = \frac{MAE_{M_1} - MAE_{M_2}}{Lat_{avg}}$, namely the difference between the Mean Absolute Error (MAE) of model M_1 (MAE_{M_1}) and the MAE of model M_2 (MAE_{M_2}) normalized by the average value of the delivery latency in the test set data (Lat_{avg}). The NAE is a scale-free metric that we deem more informative than a simple comparison between the MAEs of M_1 and M_2. In fact, small differences between the MAEs are irrelevant if the delivery latencies are, on average, large, whereas, small differences between the MAEs are relevant if, on average, delivery latencies are also small.

Finally, to assess the models' accuracy we use 60% of the available data to build the model during the training phase and the remaining 40% as test data.

3.3.1 What features to use?

A crucial challenge that has to be faced for accurately predicting the performance of any complex system via machine learning techniques is to carefully identify the set of metrics/context information to be used as input variables for the model construction (Guyon & Elisseeff 2003).

One of the first problems that we had to address while building this system was related to the difficulty to identify an optimal time window for computing the moving averages concerning the percentage of utilization of CPU and network resources. These experimental results accentuated the fact that the choice of the wrong time window could significantly affect the machine learners' accuracy. This phenomenon is clearly highlighted by the first two rows in Table 3.2, which compares the accuracy of the predictions when using specific features when the Token algorithm is being used to disseminate messages in a group of 4 machines. The fact that the Synth workload is stable over relatively long periods of time explains the 25% decrease in accuracy (measured through the NAE) when using a time window of 6 msec rather than 50 msec. On the other hand, an opposite result is obtained when considering the RBTree workload, where the accuracy decreases by 32% when using 500 msec, rather than 10 msec, time windows. This can be explained considering that shorter time windows are more sensitive to transient burstiness phenomena; this can be beneficial in presence of highly variable workloads, such as RBtree, but disadvantageous in the case of more stable workloads, such as for Synth. These results have led us to the choice of computing the moving averages across multiple time windows, ranging from 2 up to 500 msecs, and of relying on feature selection phase to filter out the ones that turned out to be uninformative or misleading for the machine learner.

Another interested finding highlighted in Table 3.2 is related to the relevance of the GC related metrics. The third row of the table reports a degradation of the model's accuracy of 29% for the Synth workload when the metric that indicates the time elapsed since the last garbage collection cycle (tLGC in Table 3.1) is not used (and otherwise using the same set of features).

These experiments have also highlighted the usefulness of incorporating time series information in the set of features used by the machine learners. To this end we pre-process the training data generated by the monitoring layer in order to include, in the set of features provided to the machine learner, the latencies of the last k TO broadcasts self-delivered by that node. As shown by the 4th row of table 3.2, when time series information is not employed, the Cubist's© predictions accuracy increases by 81% in the Synth workload scenario. This is due to the fact that, especially in the less fluctuating workloads, there is often a significant correlation among the delivery latencies of recently broadcast messages.

Finally, the last row in Table 3.2 reports a 61% decrease in the accuracy for the RBTree workload when the machine learner is not provided with information concerning the elapsed time since the token was owned by the node for the last time (toTime in Table 3.1). In the token-based algorithm, in fact, the delivery latency is strongly affected by the time elapsed before the token is owned by the sending node, and the latter is closely correlated with toTime. Overall, these results confirm the relevance of the presented semi-opaque monitoring approach, which provides the TOB layers' developers with standard interfaces to instruct the monitoring layer to trace protocol-dependant state information.

Based on the above analysis, we identified a total of 53 relevant features (43 of which being directly traced by this monitoring layer, and 10 additional ones obtained by building a time serie on the delivery latency). When faced with such an abundance (and redundancy) of available metrics, it is easy to fall prey of the, so called, curse of dimensionality (Mitchell 1997). As the number of dimensions in the feature space (i.e. the degrees of freedom of the model to be built by the machine learner) increases, in fact, the amount of training data required to ensure an equivalently dense sampling coverage of the feature space grows exponentially. This makes the machine learners much more exposed to the risk of overfitting (Dietterich 1995), a phenomenon in which the machine learner infers erroneous dependencies among random features of the training data with no causal relation to the target function, with the result of increasing their accuracy in fitting known data (hindsight) while actually degrading the accuracy in predicting new data (foresight).

Perhaps unsurprisingly, also in this experimental study the problem of overfitting evidently manifested itself when using the whole set of 53 features as input for the evaluated machine learners. More in detail, when using the whole set of features, we observed an increase of the RAE on average of around 26%, with peaks of up to 97% (for the case of Synth, 4 machines, sequencer-based algorithm), with respect to the case in which one relies on a dedicated feature selection to identify the subset of features maximizing the accuracy of the machine learner.

A possible solution to this issue would be identifying a fixed subset of features that could guarantee a good accuracy level across the whole set of workloads. Unfortunately, the experimental data demonstrates that, in practice, this is not the case. Specifically, we compared the accuracy of Cubist's models for each workload, TO algorithm and node (recall that each node builds its own performance model), using 200 different subsets of the original set of 53 features. For each of these 200 different features' subsets, we computed the average MAE across the whole spectrum of evaluated workloads/algorithms, and accordingly ranked them, thus identifying the set of features, which we call *globBest*, ensuring the minimum average MAE. For each workload and algorithm, we also identified the subset of features yielding the locally minimum MAE, which we call *locBest*. Figure 3.2 shows, for each workload, the percentage of increase of MAE (averaged across the sequencer-based and the token-based algorithm) when using *globBest* rather than the corresponding *locBest*, clearly showing that, unfortunately, no-one-size-fits-all solution exists and highlighting the necessity of running a case-by-case feature selection in order to maximize prediction's accuracy.

Note that the feature selection problem is of combinatorial nature, as identifying optimal so-

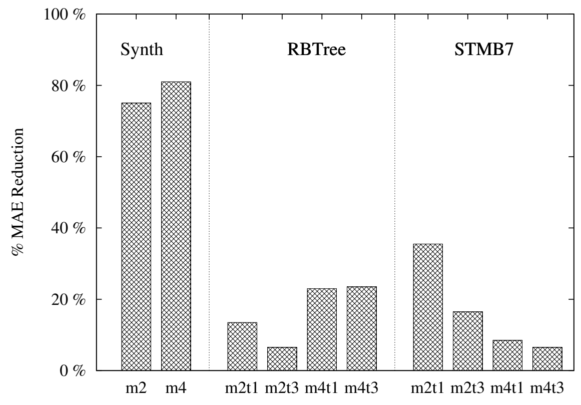


Figure 3.2: Percentage of MAE Reduction when using the features minimizing the average MAE across all workloads, rather than the features minimizing the MAE for the considered scenario.

	Sequencer							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	COR	RAE	COR	RAE	COR	RAE	COR	RAE
Synth	1.00	0.01	-	-	1.00	0.20	-	-
RBTree	0.44	0.42	0.63	0.39	0.69	0.37	0.77	0.42
STMB7	0.44	0.37	0.64	0.35	0.67	0.35	0.54	0.36

	Token							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	COR	RAE	COR	RAE	COR	RAE	COR	RAE
Synth	1.00	0.01	-	-	1.00	0.01	-	-
RBTree	0.80	0.31	0.95	0.12	0.57	0.49	0.84	0.30
STMB7	0.65	0.38	0.71	0.36	0.49	0.51	0.74	0.42

Table 3.3: Correlation Coefficient (COR) and Relative Absolute Error (RAE) of Cubist using Forward Selection.

lutions entails exhaustive searching the powerset of the feature set. This motivated the design of a number of alternative heuristic approaches that enhance efficiency at the cost of not achieving optimality. In the machine learning literature, greedy algorithms are probably the most used for implementing feature selection. There are two variants of this approach: forward selection (FS) and backward elimination (BE). In FS, features are progressively added to build larger models, whereas in BE one starts with the set of all features and progressively eliminates the least promising ones. At each iteration, one feature is added/removed and cross-validation is used to identify the best performing subsets of features. Both approaches stop when adding/removing one more feature to/from the remaining set of features no longer improves accuracy.

We report in Table 3.3 the Relative Absolute Error and correlation coefficient achieved by using the FS heuristic across all the considered workloads (we omit report results for BE as they are extremely close to those achieved by FS). The plots are relative to scenarios where the number of machines varies between 2 and 4, and the number of threads in the RBtree and STMBench7 benchmarks vary from 1 to 3. Results show that the prediction accuracy clearly depends on the complexity of the considered workload. When considering the Synth workload, the accuracy and correlation of the predictor output are extremely high, even if this workload is highly heterogeneous and encompasses phases where nodes generate both very low and high network traffic. This happens because, in each phase, the fluctuations of the delivery latencies are rather limited. The reasons for the observed stability in each phase are twofold. First, nodes are very lightly loaded, not running any computational or memory intensive tasks. Second, nodes send messages at the same rate, which makes the performance of the GCS in each phase rather stable.

The other two considered workloads are, on the other hand, definitely more challenging. First, the applications lack a well defined traffic pattern and second, nodes execute computational intensive applications. Together, these factors induce a strong variance in the self-delivery latencies. As a result, even though the set of features selected by the FS algorithm varies significantly for each workload, the performance of the predictors is similar across the two workloads, with an average correlation factor of 66% and an average RAE of 37%.

The results collected with these two workloads show an interesting trend, namely, the correlation generally grows, on average, from 59% to 73% when moving from scenarios with one thread to scenarios with three threads. This correlation is mainly due to the fact that the undelivMsgs feature (which captures the number of threads that are blocked waiting for the self-delivery of a message, see Table 3.1) becomes extremely useful in this context. Indirectly, this metric provides a measure of congestion in the system. On the other hand, this feature is only meaningful when there are at least two application level threads, as it is constantly equal to 0 in case there is a single application level thread.

3.3.2 Boosting Feature Selection

Unfortunately, despite being significantly more efficient than exhaustive searches, the FS and BE heuristics still demand the construction of hundreds of models (see the first two columns from left of Table 3.4) and require execution times on the order of the hundreds of seconds even

	FS	**BE**	**OSH**	**COM**
# Models Built	401	484	72	7
Time (sec)	250	579	53	2.1

Table 3.4: Average Execution Time of Feature Selection Algorithms.

on a fast (at the time of writing) machine equipped with two quad-core 2.33 Ghz Intel Xeon processors, 4 GB of RAM and running Linux 2.6.27[1]. These costs are clearly prohibitive in scenarios where models may have to be re-built rapidly to adapt to workload fluctuations.

To tackle this issue, we propose and evaluate the following two techniques:

- *OSH:* An Optimized Search Heuristic (OSH) that evaluates only combinations of features that were pre-selected based on a preliminary exhaustive experimentation across the whole spectrum of workloads with classical statistical tools, such as Primary Component Analysis (Pearson 1901), and cross-validation testing. This preliminary phase allowed us to identify and discard the combinations of features whose usage either provided negligible increases, or even deterioration of the prediction accuracy.
  OSH explores a total of 72 different models built by using as input features, a common set of attributes (namely, msg_size, TCPqueue, and undelivMsgS) and the combinations obtained by picking exactly one item from the following sets:
  - T={latency of the last TO broadcast, latencies of the last 5 TO broadcasts, latencies of the last 10 TO broadcasts};
  - M={no memory information, freeMem, freeMem and tLGC, freeMem and pLGC};
  - R={moving_avgs$_x$}, where moving_avgs$_x$ denotes the following set of metrics {bytesUP$_x$, bytesDOWN$_x$, TOBUp$_x$, TOBDown$_x$, totCPU$_x$, esCPU$_x$} computed over the same time window of duration x msecs, and $x \in$ {2,6,10,50,100,500} msecs.
- *Combination:* An ensemble of independent models built over largely non-overlapping sets of features and whose predictions are reconciliated on the basis of their estimated confidence interval. The intuition underlying this approach is that models built using diverse set of features have the potentiality to capture distinct phenomena affecting the delivery

[1]Interestingly, the average performance of BE is significantly worse (by a factor 2.3) with respect to that of FS even though the latter builds, on average, only 20% less models than than the former. This is due to the fact that the models built by BE have, on average, a larger number of features with respect to those explored by FS, and that the time taken by Cubist to build a model is strongly affected by the number of features it uses.

	Sequencer							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	OSH	COM	OSH	COM	OSH	COM	OSH	COM
Synth	0.2%	0.7%	-	-	2.5%	14.2%	-	-
RBTree	0.2%	3.4%	1.8%	20%	0.2%	14.2%	5.3%	14.1%
STMB7	0.4%	0.7%	2.1%	19.1%	0.3%	3.9%	3.1%	18.3%

	Token							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	OSH	COM	OSH	COM	OSH	COM	OSH	COM
Synth	7.4%	7.5%	-	-	10.7%	10.6%	-	-
RBTree	3.8%	4.1%	1.6%	2.5%	7.2%	11.7%	5.4%	8.3%
STMB7	6.4%	15.3%	11.1%	17.6%	2.6%	3.6%	20.4%	22.9%

Table 3.5: Normalized Additional Mean Absolute (NAE) Error using OSH and Combination.

latency of TOB algorithms with different degrees of accuracy. In addition, by focusing each model on a smaller subset of attributes, they are less prone to suffer of overfitting problems. Further, by selecting the prediction generated by the model with the highest degree of confidence in the current region of the feature space, the presented combination technique may enhance the accuracy of each independent model.

This combination technique generates seven models, where each model uses the same common set of attributes as in OSH (msg_size, TCPqueue, and undelivMsgS), but differs from the other ones as it uses either i) a time-series containing the last k (where we set $k = 10$) TOB latencies, or ii) moving_avgs$_x$ computed as before. In preliminary experiments we have evaluated several alternative methods for conciliating the predictions provided by the various models. We only report results for the best performing strategy, which is based on the simple approach of selecting the prediction associated with the smallest confidence interval.

As expected, by evaluating a much smaller number of models, OSH and Combination achieve striking performance gains, reducing feature selection time up to two orders of magnitude (see Table 3.4).

On the other hand, Table 3.5 reports data quantifying to what extent the quality of the predictions deteriorate when using OSH and Combination with respect to the case in which FS is used. The accuracy of OSH is extremely close to that of FS, being its average NAE around 2.2%. Concerning Combination, the average NAE increase is larger, namely around 10%. We

# Machines	# Threads	Overhead (%)
2	1	5.71
2	3	5.21
4	1	2.63
4	3	2.23

Table 3.6: Average overhead due to Monitoring Layer

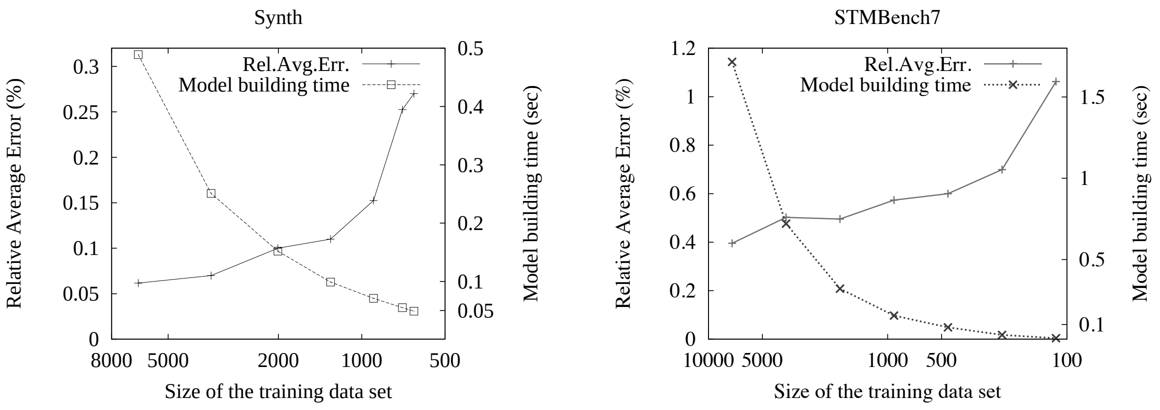


Figure 3.3: Evaluating model's building time and accuracy while varying the training data set's size.

argue that, in practical settings, this (limited) degradation of the prediction accuracy is largely compensated by the significant performance gains it achieves. On the other hand, in these experiments, we relied on Cubist's estimates of the confidence intervals, whose details are unfortunately not publicly available. An interesting open research question is whether the accuracy of the Combination technique could be enhanced by leveraging on alternative techniques, e.g. (Jiang, Zhang, & Cai 2008), for the computation of the predictions' confidence intervals.

3.3.3 Additional Performance Considerations

In Table 3.6 we report the average overhead (measured in terms of TOB throughput reduction) due to the tracing activities carried out by the Monitoring Layer with a different number of machines/threads. The numbers show that the overhead is in practice very limited, being always less than 5%, and decreasing to 2% in the case of four machines. This can be explained by considering that, as the number of nodes in the system increases, the TO delivery latency also grows accordingly. In a closed model, such as the one characterizing both the RBTree and the STMBench7 benchmarks, this leads to a reduction of the frequency of TO broadcast issued by each node and, consequently, of the frequency of messages traced by the Monitoring Layer.

In Figure 3.3 we analyze to what extent the size of the training data set affects the model's prediction accuracy and building time. We considered the model using the features selected by the FS technique for the STMBench7 and Synth benchmarks, and progressively reduced the size of the training data set. The data shown in the plots is obtained by averaging the model's accuracy and building time across the whole set of configurations (number of machines/threads and considered TOB protocol) evaluated for these benchmarks (see Section 3.2).

The plots highlight a somewhat expectable, but relevant trade-off: the model building time can be significantly reduced by using smaller training data sets, at the cost of a degradation of the predictions' quality. Specifically, the experimental data show that a very similar prediction accuracy (1%, resp. 10%, higher RAE for the Synth, resp. STMBench7) could have been achieved using 50% smaller data sets, boosting the training phase by a factor larger than 2. The selection of the training data set size represents, in fact, a key tuning knob that can be used to further reduce the time required to derive the TOB performance prediction model. Unfortunately, as also confirmed by the experimental data, the optimal choice of the training data set size is highly workload dependent and is a time consuming process which is typically performed offline.

3.3.4 Alternative Machine Learners

Finally, we compare the performance and accuracy of the models obtained by using Cubist with those generated by two other machine learning algorithms available in Weka, namely a Multilayer Neural Nework (Neural) and a Support Vector Machine regression (SMO) method (see Section 3.2.1 for an overview of these techniques). The reported results are obtained using the same set of features as input for all the machine learners, namely those selected by running the FS scheme with Cubist.

Figure 3.4 reports the NAE between Neural and Cubist, and between SMO and Cubist, averaged across all the three considered workload. By the plot, we see that Cubist significantly outperforms both Neural and SMO across almost every workload with the exception of the scenarios where sequencer-based protocol is evaluated using 2 machines. In these cases, the Weka's machine learners in fact achieve a lower Minimum Absolute Error with respect to Cubist, determining an inversion of the trend highlighted by the plot. It is interesting to note, however, that in these scenarios the correlation of Neural and SMO (not shown in the plots) is significantly

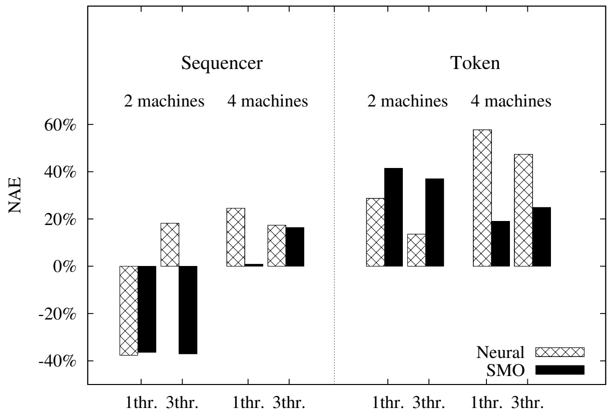


Figure 3.4: NAE of Neural and SMO with respect to Cubist.

worse (around the half) than that of Cubist. These differences can be motivated by considering that different machine learning approaches are known to optimize distinct metrics (Mitchell 1997) and by referring to the well-known "No free lunch theorem" (Wolpert 1996), which states that the performance of no single machine learner can be optimal across all possible scenarios.

As a final remark, we compared the training time of the various machine learning tools when using data set containing approximately 7.200 training cases. The results are strongly in favour of Cubist, which on average takes 0.64 seconds to build a model, whereas the average time for completing the training phase for Neural and SMO was, respectively, 243 and 575 seconds. Albeit the performance difference is quite striking, it is not completely surprising considering that Cubist is a commercial, and highly optimized performance tool (written entirely in C and parallelized to take advantage of multi-core CPUs), whereas Weka is an open-source framework designed to simplify development and testing of novel machine learning methods rather than fine-tuned for performance purposes.

3.4 Discussion

Existing performance evaluation and modelling studies of TOB (Cristian, Beijer, & Mishra 1994; Ekwall & Schiper 2007) (and related agreement problems, consensus in primis (Coccoli, Urbán, & Bondavalli 2002)) have been aimed at providing a steady state estimate of the average performance of several TOB protocols in presence of simple synthetic workloads. Typically, the purpose of these approaches is to identify the most favourable settings for each of the considered algorithmic alternatives. Also, due to the inherent complexity of TOB protocols, the only analytical models of TOB we are aware of (Coccoli, Urbán, & Bondavalli 2002; Ekwall & Schiper 2007) make rather stringent assumptions on the workload, e.g. symmetric Poissonian traffic sources generating messages at the same rate. In these works, the system model is also simplified, using synthetic (constant or exponential) communication latency distributions, that neglect important factors such as the impact of the message size on the observed latency. To the best of our knowledge, this work represents the first attempt to leverage on machine learning methods for assessing the performance of TOB protocols. Unlike existing analytical/simulation models, we leverage on statistical methods to automatically build fine-grained TOB performance models capable of forecasting in real-time the delivery latency perceived by user level applications on a per message basis.

Related works include solutions aimed at optimizing one relevant parameter of a popular class of TOB protocols: the degree of batching used in sequencer-based TOB algorithms (Friedman & Hadad 2006). The trade-off in this case is that, by using large batching values, throughput can benefit at the cost of an increase in latency. Existing works in this area have used approaches based on heuristics (Friedman & Hadad 2006), machine-learning (Romano & Leonetti 2012) and control theory (Didona, Carnevale, Galeani, & Romano 2012; Bartoli, Calabrese, Prica, Di Muro, & Montresor 2003). Unlike the work presented in this chapter, though, these solutions aim at optimizing a single parameter of a specific class of TOB algorithms. On the other hand, we investigated mechanisms aimed at predicting the absolute performance of arbitrary TOB protocols.

The work presented in this chapter is also related to the machine learning literature addressing performance prediction of computer systems. These include works aiming at forecasting the throughput of TCP flows (Mirza, Sommers, Barford, & Zhu 2007) and Pub-Sub systems (Garces-Erice 2009), solutions aimed at automatically classifying traffic based on semi-supervised learning techniques(Erman, Mahanti, Arlitt, Cohen, & Williamson 2007), at automatizing the allocation of resources in cloud-computing infrastructures (Xu, Zhao, Fortes, Carpenter, & Yousif 2008), and at generating software aging models to be used in the context of rejuvenation frameworks (Andrzejak & Silva 2008).

The idea of ensembling different machine learning models to enhance the performance of single predictors has been widely investigated in general contexts (Mitchell 1997), as well as applied to predict performance failures of complex systems (Zhang, Cohen, Symons, & Fox 2005). The latter work ensembles models based on the same set of features but representative of different phases of the life cycle of applications, and dynamically selects the model which better matches the current load scenario by ranking them based on the Brier score. Conversely, the combination technique presented in this chapter ensembles models built using largely disjoint sets of features capturing different aspects of the application workload with different degrees of accuracy.

Finally, this work is related with the body of literature addressing the issue of identifying the most informative attributes to be used by machine learners. In addition to the already mentioned greedy search techniques, such as Backward Elimination or Forward Selection (Guyon & Elisseeff 2003), we can mention also techniques, e.g.(Yang, Schopf, Dumitrescu, & Foster 2006), aimed at clustering highly correlated attributes for a preliminary screening of redundant metrics, or at identifying the set of attributes accounting for the greater variability in the output variable, such as Primary Component Analysis (Pearson 1901) and Projection Pursuit (Vetter & Reed 1999). Unfortunately, these methods do not provide a direct indication of the actual accuracy achievable by the machine learner and rely on a set of input parameters (e.g. the fraction of the total variability of the output variable should be accounted when evaluating the feature set) whose optimal settings may be not trivial to determine.

3.5 Summary

This chapter presents and evaluate a machine learning based approach to performance modeling of Total Order Broadcast protocols. The ability to provide fine-grained prediction on a per-message basis makes this technique an extremely useful building block for architecting self-optimizing replication schemes. An extensive experimental study comparing different machine learning methodologies and feature selection approaches was also presented. In addition, it introduced two novel heuristics that drastically reduce the execution time of the feature selection phase at the cost of a very limited loss of accuracy.

The next chapter will describe an approach that leverages on the work presented to build an autonomic replicated software transactional memory system based on certification protocols.

4 PolyCert: Self-Optimizing Replication with Certification Protocols

Replication clearly plays a role of paramount importance in STM's, as it represents the key mechanism to ensure data durability in face of unavoidable node failures. Unsurprisingly, replication algorithms employed in these isystems take inspiration from the vast literature on replication of transactional systems (Pedone, Guerraoui, & Schiper 2003; Kemme & Alonso 1998; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000; Couceiro, Romano, Carvalho, & Rodrigues 2009; Carvalho, Romano, & Rodrigues 2010).

Among the plethora of transactional replication mechanisms published in literature, over the last years, a wide body of research has highlighted that schemes based on Total Order Broadcast (TOB) (Defago, Schiper, & Urban 2004) and certification (Pedone, Guerraoui, & Schiper 2003; Kemme & Alonso 1998; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000) tend to outperform classic eager replication schemes based on distributed locking and atomic commit, which suffer from large communication overheads and are prone to thrashing due to distributed deadlocks (Gray, Helland, O'Neil, & Shasha 1996). Conversely, certification based schemes avoid any onerous replica coordination during the execution phase, running transactions locally in an optimistic fashion. The consistency of replicas (typically, 1-copy serializability (Bernstein, Hadzilacos, & Goodman 1986)) is ensured at commit- time, via a distributed certification phase that uses a single TOB to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of failures. Furthermore, unlike active-replication approaches that require the full execution of update transactions at all replicas (Schneider 1993), only one replica executes an update transaction, whereas the remaining replicas are only required to validate the transaction and to apply the resulting updates. This allows to achieve high scalability levels even in the presence of write-dominated workloads, as long as the transaction conflict rate remains moderate (Pedone, Guerraoui, & Schiper 2003).

In the design space of 1-copy serializable certification replication protocols, which represents the focus of this chapter, a decision that can have a dramatic impact on the actual efficiency and robustness of the system is related to how to address the trade-off between the size of the messages sent via the TOB primitive and the number of communication steps required during the transaction commit phase. Depending on how this trade-off is addressed, existing certification-based replication algorithms can be classified into three main categories:

- Solutions that disseminate the whole transaction's read set to all replicas, called *Non-voting* schemes, allow each replica to certify transactions locally, by sending both the read and write sets via an TOB primitive. This makes these protocols optimal in terms of communication steps, but also makes them prone to generate very large messages and to overload the Group Communication System.
- *Voting* schemes, which avoid broadcasting the read set of transactions by sending (via TOB) only the write set, thus drastically reducing the network bandwidth consumption. On the other hand, they incur into the costs of an additional coordination phase along the critical path of the transaction commit, which can hamper significantly performance.
- Approaches relying on the space efficient encoding of Bloom Filters (Bloom 1970) to implement a variant of the non-voting certification mechanism, called *Bloom Filter Certification* (BFC) (Couceiro, Romano, Carvalho, & Rodrigues 2009). Unlike voting mechanisms, BFC determines the outcome of transactions using a single TOB, generating smaller messages when compared to non-voting protocols. The probabilistic nature of the Bloom filter encoding, however, induces false positives in the certification phase, increasing the transaction abort rate.

The above protocols are designed to ensure optimal performance in different workload scenarios and, as we will show in the following, they can exhibit up to 10x differences in terms of maximum throughput.

The goal of this work is to alleviate the developers/administrators from the hard and time-consuming task of profiling the application and selecting the most suitable replication protocol for each deployment. Furthermore, a static configuration may lead to largely suboptimal configurations in presence of heterogeneous workloads. In these contexts, the employment of a single, statically chosen, replication mechanism, optimized for a specific workload type, will lead to suboptimal performance when processing the transactions that have different characteristics.

The solution presented in this chapter, which we named *Polymorphic Self-Optimizing Certification* (PolyCert), supports the simultaneous use of the three aforementioned classes of protocols, and relies on machine-learning techniques to determine, on a per transaction basis, the certification strategy to be adopted. PolyCert relies on a modular design, which encapsulates the logic associated with the on-line choice of the replication strategy into a generic oracle. We design and evaluate two alternative mechanisms to implement this oracle, based on two different parameter-free statistical learning techniques.

- An off-line technique based on regression decision trees (Quinlan 1993), that requires a preliminary, computational intensive, feature selection and training phase, but that was shown (in the previous chapter) to achieve high accuracy in forecasting the performance of Total Order Broadcast algorithms in presence of heterogeneous workloads.

- An on-line reinforcement learning technique, that uses an innovative, parameter-free variant of a very lightweight, but theoretically optimal solution (Auer, Cesa-Bianchi, & Fischer 2002) to face the exploration versus exploitation dilemma, i.e. the search for a balance between exploring the environment to find profitable actions while taking the empirically best action as often as possible.

Via an extensive experimental evaluation, based on a fully fledged system prototype and a range of heterogeneous benchmarks, we assess the effectiveness of the proposed approaches in terms of performance benefits and learning time. We show that PolyCert can achieve a significant speed-up with respect to static solutions and enhance the robustness of the system to unexpected fluctuations of the workload.

The remainder of the chapter is structured as follows. In Section 3.1 we present the architecture of the system, discussing the key implementation issues of the real-time monitoring Section 4.1 reports the results of a performance evaluation study highlighting the relevance of the addressed problem. The system architecture is presented in Section 4.2. Section 4.3 describes the functioning of PolyCert and the mechanisms employed to determine at run-time which certification strategy to use. The results of the experimental evaluation study are reported in Section 4.4. A short discussion on related systems is presented in Section 4.5. Section 4.6 concludes the chapter.

4.1 Motivations

As already mentioned in the previous section, existing certification-based solutions can be classified into three main categories:

- **Non-Voting Certification (NVC):** These solutions (Pedone, Guerraoui, & Schiper 2003; Carvalho, Romano, & Rodrigues 2010; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000) disseminate the whole read and write set using the TOB service, allowing every replica to determine, upon delivery of the corresponding message, the outcome (commit/abort) of the transaction, by running the certification procedure locally. These schemes are optimal in terms of communication steps, delivering excellent performance when used in workloads characterized by small transaction read sets. On the other hand, they exhibit very poor performance in presence of transactions reading a significant number of data items. Even worse, in these scenarios, the large network traffic generated by this protocol can saturate and disrupt the proper functioning of the Group Communication Service, leading to network partitions and false failure suspicions.
- **Voting Certification (VC):** These solutions (Kemme & Alonso 2000) disseminate exclusively via TOB the transaction write set, thus avoiding the issues incurred in by Non-voting schemes with large transaction read sets. On the down side, the transaction can only be certified at the site in which it was originated. This implies the need for an additional communication phase, executed using a Uniform Reliable Broadcast (URB) (Guerraoui & Rodrigues 2006) (a lighter communication primitive when compared to TOB), which is triggered by the replica where the transaction was originated in order to inform the remaining replicas of the final outcome of the transaction. This extra communication phase, which requires at least two communication steps, has a negative impact on the latency of the commit phase, which represents by far the dominating cost for small transactions. By introducing additional latency in the critical path of the commit phase, which needs to be run sequentially for conflicting transactions, these schemes can adversely affect the maximum throughput achievable by the platform (Schiper, Sutra, & Pedone 2010).
- **Bloom Filter Certication (BFC):** An alternative approach, denoted as Bloom Filter Certification (BFC) (Couceiro, Romano, Carvalho, & Rodrigues 2009), consists in encoding the read set of the transaction in a Bloom filter (Bloom 1970), a space-efficient

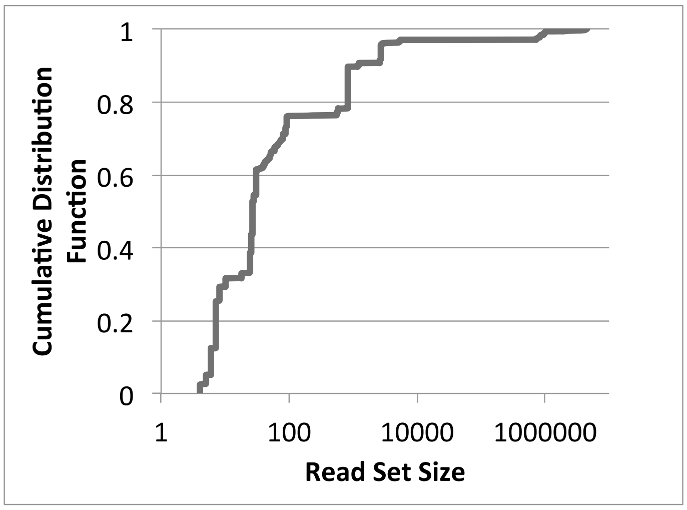


Figure 4.1: Distribution of transaction read set size (# of items read) in the STMBench7 Benchmark.

data structure that allows compressing the messages disseminated via the TOB service, while still allowing every replica in the system to deterministically certify the transactions. Unlike Voting schemes, BFC avoids additional communication steps during the commit phase. In terms of generated network traffic, even though BFC generates larger messages than voting protocols, it typically reduces significantly the size of the messages exchanged via the TOB service when compared to non-voting schemes. On the down side, BFC can suffer from false positives due to the probabilistic nature of Bloom filter-based encoding, which ultimately leads to an additional rate of aborted transactions.

From the above discussion, the performance of each of these three alternative certification mechanisms is strongly dependent on the actual distribution of the size of the read sets generated by the transactional application. Unfortunately, realistic transactional applications can exhibit very heterogeneous workloads encompassing read sets whose sizes range from less than ten to hundreds of thousands of objects. We have experimentally observed this phenomena, as illustrated in Figure 4.1, which depicts the distribution of the read set size for a widely used benchmarking application for Transactional Memories, namely the STMBench7 (Guerraoui, Kapalka, & Vitek 2007) benchmark.

In Figure 4.2 we show the results of a sensitivity analysis aimed at assessing the actual impact of the read set size distribution on the performance of the three certification schemes described above. The results were obtained using a simple synthetic benchmark adapted from

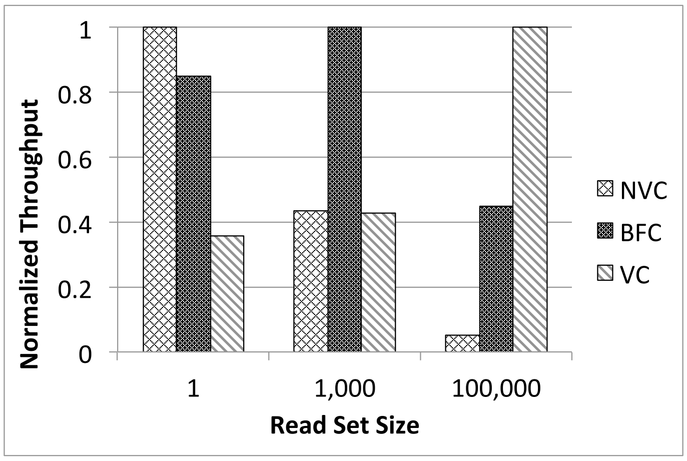


Figure 4.2: Throughput of three certification strategies with different read set sizes (# of items read).

the Bank Benchmark originally used in (Couceiro, Romano, Carvalho, & Rodrigues 2009). This benchmark simulates the concurrent transfer of funds from different bank accounts (modelled as a simple array of doubles), and was altered to vary the number of items read within a transaction in the range [1,100'000]. Further, to focus only on the effects due to variations of the read set size, which represents the goal of this sensitivity analysis, we configured the benchmark to never generate conflicts among transactions. The only aborts experienced in the system are therefore those determined by false positives with the BFC scheme (which was configured to have an additional abort-rate of 1%, as in (Couceiro, Romano, Carvalho, & Rodrigues 2009)). These results were obtained running on a cluster of eight nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet (which represents the reference experimental platform used in the remainder of the chapter). The replicated STM and the certification protocols were implemented in JAVA. The system uses two main components: i) a state of the art Software Transactional Memory (STM), namely JVSTM (Cachopo & Rito-Silva 2006), used to manage local concurrency, and ii) a replicated key/value store, used to maintain associations between unique object identifiers and object instances. Further details on the system architecture will be provided in Section 4.2.

These experimental results highlight that no-one-fits-all solution exists that maximizes the throughput across all the considered workloads. On the contrary, NVC provides the best performance in the scenario with small read sets, BFC in the scenario with 1000 items in the read

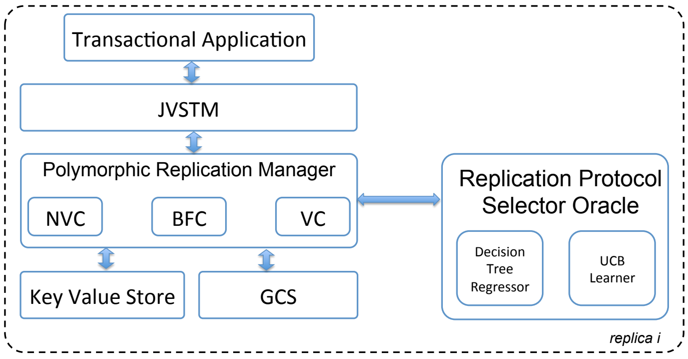

Transactional Application
JVSTM
Polymorphic Replication Manager
NVC
BFC
VC
Key Value Store
GCS
Replication Protocol Selector Oracle
Decision Tree Regressor
UCB Learner
replica i

Figure 4.3: PolyCert's Architectural Overview (Single Node Perspective).

set, and VC is by far the best performing protocol with large read sets. Further, the relative difference in the performance between the best and worst performing protocol for each scenario ranges from a factor 2.5x (BFC vs VC, read set size equal to 1000) to 10x (VC vs NVC, read set size equal to 100'000).

4.2 PolyCert Architecture

PolyCert's architecture is depicted in the diagram shown in Figure 4.3. At the topmost layer, it exposes the API of an object-oriented STM, which is however fully replicated across a number of distributed nodes. The API provided to applications is a transparent extension of JVSTM's API, a state-of-the-art STM relying on an efficient Multi Version Concurrency Control (MVCC) algorithm (Bernstein, Hadzilacos, & Goodman 1986); a strong advantage of JVSTM is that read-only transactions are never required to block. The JVSTM programming paradigm requires that the programmer encapsulates any shared mutable state within VBoxes, which are then managed by JVSTM's MVCC to ensure transactional atomicity and isolation. This allows separating the transactional and non-transactional state of the application, ensuring strong atomicity (Martin, Blundell, & Lewis 2006) at no additional costs. We modularly extended JVSTM by augmenting it with what we have named a Polymorphic Replication Manager (PRM).

The PRM is in charge of triggering the execution of a certification protocol for each of the locally executed transactions, and to participate in the certification of transactions that have been executed at remote nodes. A unique feature of PRM, with regard to existing replication managers, is that it is able to determine, for each locally executing transaction, which certification algorithm is more appropriate, given the characterization of the transaction. The logic for determining the certification scheme is encapsulated by the abstraction of a Replication Protocol Selector Oracle (RPSO), whose interface exports two main functionalities:

- Given a local transaction, it selects the most appropriate certification protocol to be executed by the PRM. In Section 4.3.1 we will present and evaluate two performance forecasting methods which are based on alternative machine learning techniques.

- Collecting historical data about the execution of past transaction, to improve the selection process. For this, it exports an interface that allows the PRM to register the following information: i) the commit time experienced by a transaction, and ii) the certification protocol that it used. This allows the RPSO to gather, store and analyse (either on-line or off-line) statistical data on the distribution of the commit time of transactions.

The PRM also interacts with two other components, the Group Communication System (GCS) and a local Key Value Store (KVS).

The GCS, Appia (Miranda, Pinto, & Rodrigues 2001) in our implementation, provides a number of communication abstractions required by the PRM: view synchronous membership, TOB and URB (Guerraoui & Rodrigues 2006).

Finally, the Key Value Store is a weak hash map, used to maintain the mapping between application level transactified objects (namely, containing JVSTM VBoxes) and replica-wide unique object identifiers, which are generated automatically by the framework upon creation of a new transactional object. More in detail, an entry of the key/value store contains the unique object identifier, as its key, and a *weak reference* to the local transactional object as its value. This information is used by the PRM, upon reception of a commit request for a remote transaction T, to retrieve in the local JVSTM instance the objects that were read/updated by T during its remote execution. The usage of a weak hash map ensures that the Java garbage collector is not prevented from discarding the object referenced by a hash map entry whenever all application level references to the object have been removed, thus avoiding any interference with the local JVM garbage collection mechanism.

4.3 The PolyCert replication protocol

PolyCert is designed to allow the simultaneous use of all the three TOB-based certification protocols introduced before, namely NVC, BFC and VC. Specifically, at commit time (when the size of the transaction read and write set is already known), the PRM invokes the RPSO to determine which certification protocol to use in order to finalize the transaction's execution. This clearly implies that concurrent transactions may use different protocols, which need to coexist without endangering the correctness of the system.

The three protocols lend themselves quite naturally to coexist, given that all of them rely on a first common phase during which they establish, via the TOB primitive, the global serialization order for a committing transaction (even though each protocol piggybacks different information). Naturally, certification messages need to be tagged with a label that specifies which of the protocols is being used for each transaction.

Upon delivery of an TOB message, the PRM performs the following steps:

- The message is inserted in a queue containing the transactions to be certified.
- If the transaction is being certified using NVC or BFC, no further processing is done until the message reaches the head of the queue.
- If the transaction is being certified using VC, and the node was the originator of the transaction, the following procedure is executed immediately. Upon the enqueuing of a transaction, say T, if and only if T does not conflict with any of the transactions already present in the certification queue (i.e. the read set of T does not intersect with the write sets of the transactions already present in queue), T is immediately validated, by verifying whether the snapshot that it read is still fresh, the URB convoying the decision outcome is triggered. This optimization, originally proposed in (Schiper, Sutra, & Pedone 2010), allows to overlap the URB dissemination phases of (non-conflicting) transactions with the time spent by transactions in the certification queue, reducing the risk of convoy effects which are otherwise known (and confirmed by our experience) to significantly hamper performance of VC schemes.

Subsequently, messages are removed from the head of the queue one by one and the corresponding transactions validated in a sequential manner. More specifically:

- If the message that is extracted from the head of the queue corresponds to a transaction that is being certified using NVC or BFC, each node applies locally the corresponding certification algorithm. Essentially, it verifies if the read set accessed by the transaction is still valid, and commits or aborts the transaction accordingly.
- If the message that is extracted from the head of the queue corresponds to a transaction that is being certified using VC, the node checks if a vote has already been received or not. If a vote has been received and the decision was to commit the transaction, the write set is applied. Otherwise, if the vote has been received and the decision was to abort the transaction, the write set is discarded (actually, the transaction can be immediately removed from the queue as soon as an abort vote is received). On the other hand, if a vote has not been received and the transaction is remote, the certification is suspended until the corresponding vote is received. Finally, if a vote has not been received and the transaction is local, the node validates the transaction as described above, and issues the vote (this corresponds to the scenarios where the optimization described before cannot be applied).

As a final remark, note that, since all replicas deliver the certification messages in the same order due to the use of the TOB primitive, and decide deterministically which certification protocol to use, consistency is guaranteed even in presence of concurrent transactions being processed using different certification schemes.

4.3.1 Replication Protocol Selection Oracle

As already mentioned, the Replication Protocol Selector Oracle abstraction (RPSO) is a convenient form of encapsulating different performance forecasting techniques. In this chapter, we present two implementations of the RPSO, using two different machine learning techniques, namely a regressor based on decision trees (Quinlan 1993), and a reinforcement learning technique, namely UCB (Auer, Cesa-Bianchi, & Fischer 2002), as described below.

4.3.1.1 Oracle based on regressor decision trees

In order to forecast the time necessary for committing a transaction with each of the three considered certification strategies, we start by forecasting the size, m_p, of the TOB message that would be generated by each of the certification protocols $p \in \{NVC, BFC, VC\}$. This corresponds to the number of bytes required to transmit the transaction read and write set with NVC, the transaction write set with VC, and the write set and the Bloom filter based encoding of the transaction read set with BFC.

Next, we forecast the time for marshalling and validating a transaction with each of the considered certification schemes. To this end, we maintain, for each certification strategy, a moving average of the average marshalling time per byte, denoted as T_p^m, and of the validation time, denoted as T_p^v, for all $p \in \{NVC, BFC, VC\}$. Further, for BFC, we maintain moving averages of the time required to build a Bloom filter that encodes the read set (normalized by the read set's size), denoted as T^{BF}. Finally, for VC, we maintain also the moving averages of the self-delivery latency of the URB convoying the vote from the transaction's initiator, denoted as T^{URB}. Note that the choice of measuring self-delivery latencies allows us to avoid distributed clock-synchronization mechanisms, which in preliminary experiments revealed not to be sufficiently accurate for our purposes.

Using the metrics above, the commit latency T_p for a transaction using certification protocol p is then forecast as follows:

$$T_{NVC} = T_{NVC}^m \cdot m_{NVC} + T_{NVC}^v + T^{TOB}(m_{NVC}) \tag{4.1}$$

$$T_{BFC} = T_{NVC}^m \cdot m_{BFC} + T_{VC}^v + T^{BFC} \cdot rsSize + T^{TOB}(m_{BFC}) \tag{4.2}$$

$$T_{VC} = T_{VC}^m \cdot m_{VC} + T_{VC}^v + T^{TOB}(m_{VC}) + T^{URB} \tag{4.3}$$

where $T^{TOB}(m)$ is the forecast latency for self-delivering a message of size m using the TOB primitive. To this end, we exploit the results obtained from the work presented in the previous chapter, where we evaluated a series of (off-line) machine learning techniques to forecast TOB's performance, including neural-networks (Haykin 1994) and support vector machines (Shevade, Keerthi, Bhattacharyya, & Murthy 2000). In the light of the results achieved, we integrated in PolyCert a regression technique relying on the Cubist, which we use to predict the TOB self-delivery latency (expressed in microseconds).

In order to generate the training data for the decision tree regressor we ran the synthetic benchmark described in Section 4.1, for each of the three considered certification protocols,

Metric	**Description**
freeMem	Free memory in the Java Virtual Machine
tLGC	The time since the last garbage collection
pLGC	% of time since the last GC cycle w.r.t. the time between the last 2 GC cycles
undelivMsgs	#TOB msgs and not yet self-delivered
bytesUp$_x$	#Bytes received over a x msec. time window
bytesDown$_x$	#Bytes sent over a x msec. time window
TOBUp$_x$	#TOB deliver events over a x msec. time window
TOBDown$_x$	#TOB broadcast events over a x msec. time window
totCPU$_x$	% total CPU utilization over a x msec. time window
esCPU$_x$	% CPU utilization by ES thread over a x msec. time window
TCPqueue	Outgoing messages queued at the Transport Layer

Table 4.1: List of metrics (features) collected by the Monitoring Layer.

varying every 3 minutes the read set size of the generated transactions in the set {10, 100, 1'000, 100'000}. Overall the training data set gathered by each replica is constituted, on average, by approximatively 25'000 samples, reporting the self-delivery latency for each TOB message along with the message size, and a total of 53 different metrics (i.e. context information), also referred to as features, including averages on multiple time scales and time series of a plethora of metrics (listed in Table 4.1) concerning the utilization of various system resources (CPU, RAM and Network). The choice of this synthetic benchmark to generate the training data set has the following rationale: since this benchmark generates transactions with very heterogeneous read set sizes, it allows gathering a good a-priori knowledge on the performance of a wide range of possible workload scenarios that the system may face when running more complex, realistic applications.

To minimize the effects of overfitting, which are likely to occur given the high dimensionality of the feature space, we run a greedy feature selection algorithm (Forward Selection (Guyon & Elisseeff 2003)) aimed at discarding loosely correlated features and boosting the predictor's accuracy. Feature selection is by far the most time consuming phase of the off-line training, taking on average 45 minutes (per replica) when run on a PC equipped with Intel Core 2 CPU with a 2.2GHz clock-rate and 2GB of RAM. On the other hand, feature selection allows to achieve a significant improvement in the accuracy of the predictions, as highlighted by the results shown in Table 4.2, which report the correlation factor and mean absolute error using 10-fold cross-validation before and after performing feature selection.

Metric	**Before FS**	**After FS**
Relative Absolute Error	0.81	0.30
Correlation Coefficient	0.17	0.76

Table 4.2: Prediction accuracy of the decision tree regressor before and after feature selection.

4.3.1.2 Oracle based on the UCB online learner

The second implementation of the Oracle Layer employs an on-line learning technique. Therefore, it does not require an a-priori computational intensive off-line training. Instead, it relies on a lightweight reinforcement learning (RL) technique that updates the knowledge of the oracle as the system is running.

This oracle relies on a customized, self-tuning version of a state of the art RL algorithm, called UCB (Upper Confidence Bounds), which solves (in a theoretically optimal manner) a classical on-line learning problem, known in literature as the *multi-armed bandit* (Robbins 1952). In this problem, a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy in order to maximize the average reward.

Formally, each arm i of the bandit, for $0 \leq i \leq k$, is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal of the agent is to learn which arm i maximizes the criterion:

$$\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$$

that is, achieves maximum average reward. To this purpose, the learning algorithm needs to try different arms in order to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied that minimize the *regret*, defined as

$$\mu^* n - \mu_i \sum_{i=1}^{K} E[T_i(n)]$$

where $T_i(n)$ is the number of times arm i has been chosen. Building on the idea of *confidence bounds*, the UCB algorithm creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of *optimism in the face of uncertainty*, the algorithm picks the option with the highest current bound. Interestingly, this allows UCB to achieve a logarithmic bound on the regret value not only asymptotically, but also for any finite sequence of trials (Auer, Cesa-Bianchi, & Fischer 2002).

More in detail, UCB assumes that rewards are distributed in the [0,1] interval, and associates each arm with a value:

$$\overline{\mu_i} = \overline{x_i} + \sqrt{2\frac{\log n}{n_i}} \tag{4.4}$$

where $\overline{x_i}$ is the current estimated reward for arm i, n is the number of the current trial, n_i is the number of times the level i has been tried.

The right-hand part of the sum in Eq. 4.4 is an *upper confidence bound* that decreases as more information on each option is acquired. By choosing, at any time, the option with maximum $\overline{\mu_i}$, the algorithm searches for the option with the highest reward, while minimizing the *regret* along the way.

In order to apply the UCB technique to our problem, we had two face two main issues, which we discuss in the following paragraphs.

State space discretization As we have illustrated in Section 4.1, the performance of certification depends on the workload characterization. Thus, using a single UCB instance, having as arms the three alternative protocols for all possible scenarios is clearly not a viable solution. This observation raises the problem of discretizing the workload state space into a set of distinct, representative, classes of workload scenarios. This allows, in fact, to associate a different instance of a UCB learner with each discretized interval of the workload's parameter space, and to train each instance to choose among the three considered protocols under specific workload conditions.

The discretization process involves a delicate trade-off: a finer (i.e. denser) discretization can lead, eventually, to more accurate predictions across the entire state space, but requires the training of a larger number of UCB instances, which can lead to a considerable increase of the learning time. In order to determine an appropriate discretization strategy, we analysed the average commit latency of each of the three protocols as a function of the read set size using the synthetic benchmark introduced in Section 4.1. The results, reported in the log-log plot of

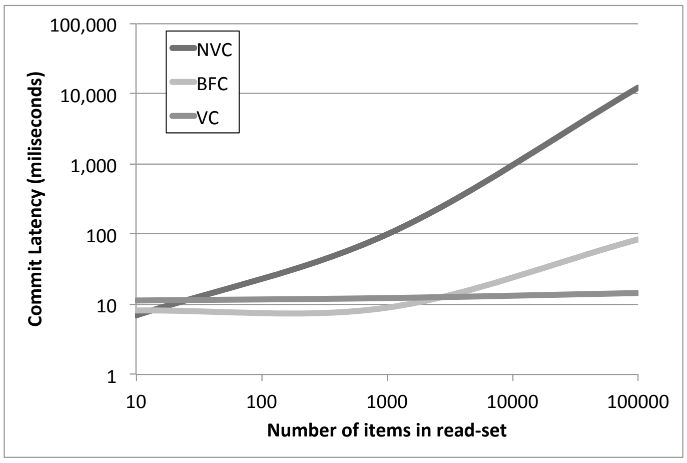


Figure 4.4: Commit latency as a function of the read set size.

Figure 4.4, highlight that, for NVC and BFC, the read set size and commit latency exhibit a heavy-tail relationship. At the light of this observation, we opted to use the read set size as the reference variable to discriminate different workload situations, and we discretized it using exponentially increasing intervals, where each sampling interval is defined by the range [10^i, $10^i + 1$] with $i \in \{1 \ldots 6\}$). This choice allowed us to partition the state space into a small number of intervals, thus reducing learning time, while associating each discretized interval with fluctuations of approximately the same relative amplitude in the commit latency, even for the case of the NVC, whose commit latency is the most sensible to variations of the read set size.

Definition of the reward function UCB is based on the assumption that rewards are distributed in the [0,1] interval, whereas, as we have seen in Figure 4.4, the commit latencies are distributed over a very large domain. This required defining a mapping function, denoted as $R(t)$, taking as input a commit latency, t, and outputting a value (the reward) distributed in the [0,1] interval. In order to preserve the relative distance among samples before and after applying the mapping function we employed the following linear transformation:

$$R(t) = \frac{maxLatency - min\{maxLatency, t\}}{maxLatency}$$

which relies on the parameter $maxLatency$, defining a threshold for the commit latency, above which the reward is mapped to the value 0. Based on preliminary experiments, we observed that the correct definition of the $maxLatency$ parameter value has a fundamental impact on the effectiveness of UCB: excessively low or high values would in fact lead to saturating the reward function, preventing UCB to distinguish sensibly the performance of the various protocols. Also, the manual tuning of this parameter is an extremely time-consuming task, given that the setting of *maxLatency* was found to depend strongly on the characteristics of the user level application. For instance, we noted that, when testing this approach with the STMBench7 benchmark, we had to increase the value of *maxLatency* by a factor approximately 27x larger than when using the synthetic benchmark described in Section 4.1.

This led us to design a self-tuning mechanism to define the value of the *maxLatency* parameter. This mechanism is based on the observation that the (average) commit latency when using VC is i) largely unaffected by the read set size (given that it does not disseminate the read-set), and ii) lower than that of *both* NVC and BFC for sizes of the read set larger than some threshold (this threshold being unknown and dependant on the application and deployment scenario). In other words, VC's commit latency represents a consistent upper bound for NVC's and BFC's commit latencies below a given read set's size threshold, in which the two protocols typically exhibit alternate performances. On the other hand, it represents a lower bound for NVC's and BFC's commit latency for high read set's size, a scenario in which it is actually unnecessary to be able to accurately predict their performance, given that VC outperforms them significantly.

This makes the VC's average commit latency, denoted as T_{VC}, a good reference point for UCB's *maxLatency* parameter value. This insight led us to define the following rule:

$$maxLatency = \overline{T_{VC}} \cdot (1 + \sigma(T_{VC}))$$

where $\sigma(T_{VC})$ denotes the standard deviation (more precisely the squared root of the sampling variance) of T_{VC}. In order to instantiate this formula, upon boot-strapping of the system, we execute transactions using the VC scheme until the following stopping condition is reached:

$$\sigma(T_{VC}) < 2 \cdot \overline{T_{VC}}$$

which in our experiments typically implied a few tens of transactions (and that was however upper bounded to 100 transactions to ensure robustness in the presence of highly disperse sampling data). To minimize the impact of this (typically quite short) bootstrapping phase on the learning time, we provide the observed sampling data to the corresponding UCB's instances also during this phase (in which UCB's instances are not being queried to choose the replication protocol), thus allowing them to gather statistical information concerning the reward of the arm associated with the VC protocol.

A further optimization that we designed in order to minimize learning time is to have the replicas periodically exchange and merge the locally gathered statistical information concerning the reward distributions of UCB's arms. This allows the replicas to mutually benefit from the statistical knowledge that they have gathered so far, narrowing the upper confidence bounds of the UCB's instances and accelerating their convergence. To minimize the overhead, we piggyback periodically (e.g. each 10 seconds in our experiments) the state of the 6 UCB's instances maintained at each replica (encoded by the tuple $< \overline{x_i}, n_i, n >$ for each of its three arms $i \in \{NVC, BFC, VC\}$, and globally accounting to around 100 bytes) to the TOB messages generated by the PRM. As soon as updated statistical information from a different replica is received, the information concerning the local UCB instances is updated by setting, for each arm i:

- the value of $\overline{x_i}$ to the average of the local and remote values of $\overline{x_i}$, weighted proportionally to the number of times i was played locally and remotely, namely:

$$\overline{x_i} = w_i^{loc}\overline{x_i} + w_i^{rem}\overline{x_i^{rem}}, \text{ where } w_i^{loc} = \frac{n_i}{n_i + n_i^{rem}} \text{ and } w_i^{rem} = 1 - w_i^{loc}$$

- the value of n_i and n to the sum of their, respectively, local and remote values, namely $n_i = n_i + n_i^{rem}$ and $n = n + n^{rem}$

4.4 Experimental Evaluation

In this section we report the results of an experimental study aimed at assessing the performance gains achievable by PolyCert, and the adequacy of the proposed machine-learning based self-optimizing mechanisms.

We start by considering the synthetic benchmark already used in Section 4.1 that, thanks to its simplicity and predictability, allows us to analyse the performance of PolyCert in precisely identifiable workload scenarios. We then evaluate STMBench7, already presented in Section 3.2.2 of the previous chapter. STMBench7 is a complex benchmark that features a number of operations with different levels of complexity over an object-graph with millions of objects,

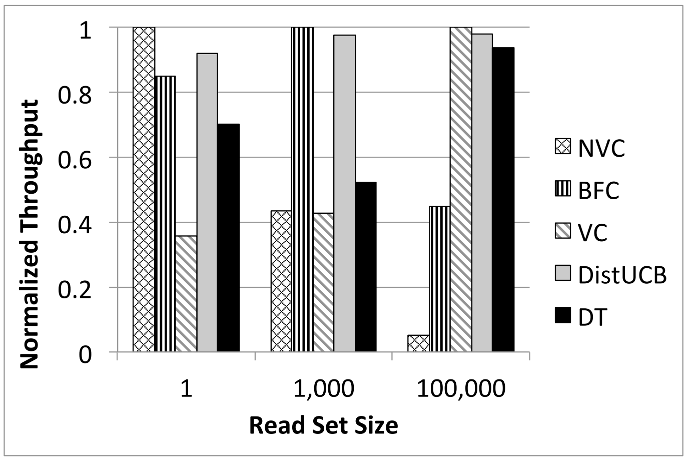


Figure 4.5: Normalized throughput of the adaptive and non-adaptive protocols (Bank benchmark).

generating a very intense and heterogeneous workload for the GCS. All the throughput results reported in the following were obtained averaging over a number of runs sufficient to ensure that the width of the 90% confidence intervals for the throughput was less than 10% of the corresponding average value.

The bar plot in Figure 4.5 reports the normalized throughput (with respect to the optimal non-adaptive workload) for each of the workloads generated by the Bank benchmark, including the versions of PolyCert when employing the oracles based on regressor decision trees and UCB (respectively dubbed as DT and DistUCB in the plot). These experiments were run by switching the workload every three minutes, thus the reported performance incorporates also data gathered during the initial phases during which we bootstrap the statistical information of UCB.

The presented experimental data shows that the on-line learning oracle using UCB (with the optimization for periodically exchanging statistical information among replicas enabled) achieves a performance very close to the corresponding optimal protocol for each scenario, namely on average around 5% less than the optimal solution and in the worst case, the scenario where the transaction's read set size is set equal to 1, less than 10% from the optimum. In this scenario, UCB alternates between BFC and NVC, whose performances are quite close (differing by around 15%); in several runs some replicas eventually converged towards the choice of BFC. In all the remaining scenarios, after a short bootstrapping phase, the replicas converged consistently towards the choice of the optimal certification protocols, which explains why they achieved performance almost indistinguishable from those of an optimally tuned non-adaptive protocol.

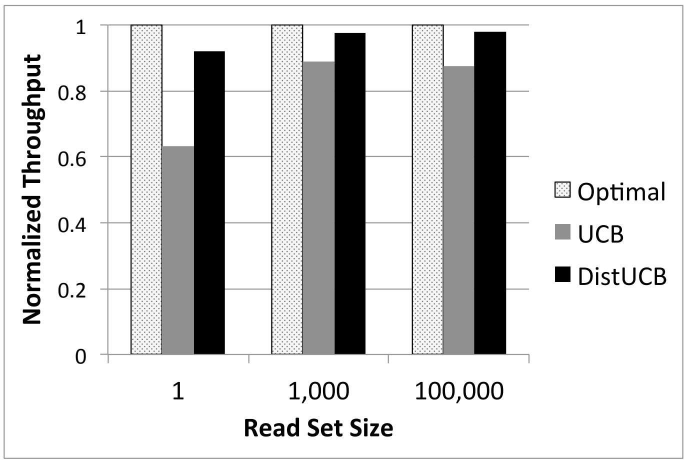


Figure 4.6: Normalized throughput of UCB and DistUCB over a three minute run (Bank benchmark).

On the other hand, the performance achieved by the oracle based on regressor decision trees was significantly worse. When using DT, the performance of PolyCert was approximately 25% worse than that of the corresponding optimal non-adaptive scheme (across the three workloads). Note that, DT was still able to outperform the second best non-adaptive protocol (but not the optimal choice). A main source of inefficiency in the implementation of the DT oracle is the following: it relies on the Java Native Interface (JNI) to query the decision tree-based model generated by Cubist, implemented in C. The overheads due to JNI are negligible in the scenario with read set size equal to 100'000 items, whose transactions have a local execution time in the order of a few tens of milliseconds. On the other hand, JNI's overheads have a negative impact on performance in the scenarios with smaller read set sizes, in which transactions have a local execution time on the order of just a few tens of microseconds. The performance of DT is lower in the scenario with a read set size of 1000, as in this case the DT oracle had a lower accuracy in forecasting the TOB self-delivery time, and erroneously biased its decisions towards the voting protocol (which is chosen in approximately 30% of the cases on average).

In Figure 4.6 we contrast the performance of the UCB oracle (again in terms of normalized throughput vs the optimal non-adaptive protocol), over a three minute run, with and without enabling the optimization of exchanging periodically (each 10 seconds) statistical information among replicas to improve learning. The data clearly shows the effectiveness of this optimization, with speed-ups larger than 25% due to the fastest convergence towards the optimal non-adaptive solution. Figure 4.7 provides more detailed insights on the speed of convergence of UCB and

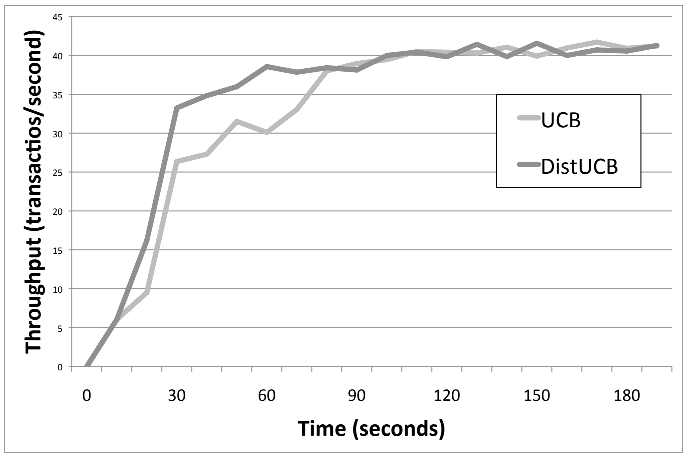


Figure 4.7: Evolution of throughput over time with UCB and DistUCB (Bank benchmark - 100'000 read set size scenario).

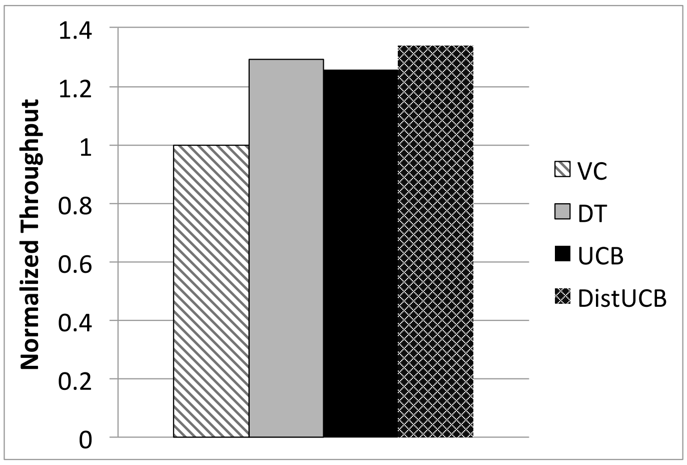


Figure 4.8: Normalized throughput of the adaptive and VC protocols. NVC and BFC not reported as they caused the collapse of the GCS layer. (STMBench7 benchmark)

DistUCB versus the optimal solution, reporting the average throughput over 10 seconds time windows, achieved by the two protocols. The plots clearly highlight the positive effects, in terms of learning time reduction, due to the exchange of statistical information occurring, in particular, at the time instants 10, 20 and 30 (seconds), that nearly halves the time required to converge to the optimal choice.

Finally, we assess the performance of PolyCert with STMBench7, plotting the corresponding results in Figure 4.8. The benchmark was configured to use the write-dominated workload with long traversals, which generates approximately 90% of update transactions, thus allowing us to focus on the performance of the transactions that require the activation of a commit-time certification phase. As shown in Figure 4.1, around 5% of transactions (namely the so-called *long traversal* transactions) in this benchmark have read set sizes larger than 500'000 items. As a consequence, when using either NVC or BFC, this benchmark generates a very high traffic volume that, in all our runs, eventually determined the saturation and the collapse of the GCS. This is the reason why in Figure 4.8 we only report the throughput of VC, DT, UCB and DistUCB (normalized with respect to the throughput of the optimal non-adaptive protocol, namely VC). In this scenario, the adaptive protocols clearly outperform the non-adaptive VC scheme, thanks to their ability to use the more efficient NVC and BFC protocols to handle transactions with smaller read set's size. The speed-up of PolyCert when using the three alternative oracles ranges from 25% to 35%, with the best performance also in this case achieved by DistUCB.

Overall, this experimental data demonstrated the effectiveness and viability of the proposed self-tuning polymorphic replication technique. The reported results highlight in particular the efficiency of the DistUCB oracle, which, not needing any time-consuming off-line training phases, and being totally parameter-free, results as extremely convenient for deployment in real-life practical scenarios. Interestingly, PolyCert does not only provide benefits in terms of performance, but also in terms of robustness, avoiding to saturate the GCS in presence of transactions with extremely large read sets, a main source of instability for BFC and, in particular, NVC.

4.5 Discussion

The work presented in this chapter is clearly related to the vast literature on replication of transactional systems, and in particular to the more recent works relying on TOB to achieve a replica-wide agreement on the transaction serialization order (Kemme & Alonso 1998; Pedone, Guerraoui, & Schiper 2003; Kemme & Alonso 2000; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000). All these protocols adopt a single static strategy, unlike PolyCert which, not only allows for the simultaneous coexistence of multiple certification strategies, but autonomically determines, on a per-transaction basis, the most adequate replication protocol to employ using machine-learning techniques.

PolyCert is also related to the body of research on autonomic computing, and in particular to the field of self-optimizing databases. In this context, several approaches have been proposed based on the idea to automatically analyse the incoming workload, e.g. (Martin, Elnaffar, &

Wasserman 2006), to automatically identify the optimal database physical design or self-tune some of the DBMS inner management schemes, e.g. (Bruno & Chaudhuri 2007; Paiva, Ruivo, Romano, & Rodrigues 2014). However, none of these approaches investigated the issues related to autonomically adapt the replication scheme. We argue that this is mainly due to the fact that current DBMSs, because of the high complexity of their architecture, lack the flexibility required to dynamically adapt such low level mechanisms.

4.6 Summary

Since the parameter space defining the workload of transactional applications is extremely wide, it is extremely challenging to devise universal transactional replication solutions capable of guaranteeing optimal performance in any possible scenario. This chapter proposes a self-tuning adaptive scheme, named PolyCert, that allows for the simultaneous coexistence of multiple TOB-based certification schemes. PolyCert uses parameter-free machine learning techniques to determine the optimal replication strategy to use on a per-transaction basis. The self-tuning strategy of PolyCert allows to achieve significant speed-ups when compared with non-adaptive certification protocols. Furthermore, it also improves the robustness of the replicated data platform, avoiding to saturate the GCS in the presence of transactions with extremely large read sets, a main source of instability for several certification protocols.

However, since in Polycert it is only possible to use protocols from the same family, namely certification based replication protocols, which only differ in the way transactions are validated, the next chapter will address the more complex and generic problem of adaptive reconfiguration among arbitrary replication schemes by presenting a system that supports radically different replica consistency protocols.

5 MorphR: Framework for Automatic Adaptation of Replication Protocols

In the previous chapter we have seen how to autonomically select among multiple protocols based on the same family. However, as we show in this chapter, often the best protocols for different workloads are protocols from different families. This chapter addresses this issue by introducing MorphR, a framework supporting automatic adaptation of the replication protocols employed by STMs that is able to switch among protocols from different families.

Unfortunately, as we clearly show in this chapter, there is not a single replication strategy that outperforms all other strategies for a wide range of workloads and scales of the system. That is, the best performance of the system can only be achieved by carefully selecting the appropriate replication protocol in function of the characteristics of the infrastructure (available resources, such as number of servers, CPU and memory capacity, network bandwidth, etc) and workload characteristics (read/write ratios, probability of conflicts, etc).

These facts raise two significant challenges. First, given that both resources and workloads are dynamic, the data grid platform must support the run-time change of replication protocols in order to achieve optimal efficiency. Second, the amount of parameters affecting the performance of replication protocols is so large, that the manual specification of adaptation policies is cumbersome (or even infeasible), motivating the need for fully autonomic, self-tuning solutions.

This chapter addresses these issues by introducing MorphR, a framework supporting automatic adaptation of the replication protocols employed by replicated STM systems. The contributions of this chapter are the following:

- We present the results of an extensive performance evaluation study using an open source transactional data grid (Infinispan by Red Hat/JBoss), which we extended to support three different replication protocols, namely primary-backup (Budhiraja, Marzullo, Schneider, & Toueg 1993), distributed locking based on two-phase commit (Bernstein, Hadzilacos, & Goodman 1986), and total order broadcast based certification (Pedone, Guerraoui, &

Schiper 2003). We consider workloads originated by both synthetic and complex standard benchmarks, and deployments over platforms of different scales. The results of our study highlight that none of these protocols can ensure the best performance for all possible configurations, providing a strong argument to pursue the design of abstractions and mechanisms supporting the online reconfiguration of replication protocols.

- We introduce a framework, named MorphR, which formalizes a set of interfaces with precisely defined semantics that need to be exposed (i.e. implemented) by an arbitrary replication protocol in order to support its online reconfiguration, i.e. switching to a different protocol. The proposed framework is designed to ensure both generality, by means of a protocol-agnostic generic reconfiguration protocol, and efficiency, whenever the cost of the transition between two specific replication protocols can be minimized by taking into account their intrinsic characteristics. We demonstrate the flexibility of the proposed reconfiguration framework by showing that it can seamlessly encapsulate the three replication protocols mentioned above, via both protocol-agnostic and specialized protocol switching techniques.

- We validate the MorphR framework, by integrating it in Infinispan, which allows to assess its practicality and efficiency in realistic transactional data grids. A noteworthy result highlighted by our experiments is that the MorphR-based version of Infinispan does not incur in perceivable performance overheads in absence of reconfigurations (which is expected to be the most frequent case), with respect to the non-reconfigurable version. We use this prototype to evaluate the latencies of generic and specialized reconfiguration techniques, demonstrating that the switching can be completed with a latency in the order of a few tens of milliseconds in a cluster of 10 nodes employing commodity-hardware.

- We show how to model the problem of determining the best replication protocol given the current operational conditions as a classification problem. Via an extensive experimental study, relying on three machine learning techniques and heterogeneous workloads and platform scales, we demonstrate that this learning problem can be solved with high accuracy.

The remainder of the chapter is structured as follows. Section 5.1 reports the results of a performance evaluation study highlighting the relevance of the addressed problem. The system architecture is presented in Section 5.2 and its main components are presented in Sections 5.3 and 5.4. The experimental evaluation is reported in Section 5.5. A short discussion on related systems is presented in Section 5.6. Section 5.7 concludes the chapter.

5.1 Motivations

In the introduction above, we have stated that there is not a single replication strategy that outperforms all others. In this section, we provide the results of an experimental study backing this claim. Before presenting the experimental data, we provide detailed information on the experimental platform and on the benchmarks used in our study.

5.1.1 Experimental Platform

We used an open-source replicated STM, namely Infinispan (Marchioni 2012) by Red Hat/JBoss, as reference platform for this study. At the time of writing, Infinispan is the reference NoSQL platform and clustering technology for JBoss AS, a mainstream open source J2EE application server. From a programming API perspective, Infinispan exposes a key-value store interface enriched with transactional support. Infinispan maintains data entirely in memory, using a weakly consistent variant of a multi-version concurrency algorithm to regulate local concurrency. More in detail, the Infinispan prototype used in this work (namely, version 5.2.0), ensures two non-serializable consistency levels: repeatable read (ISO 1999), and a variant that performs an additional test, called *write-skew check*, which aborts a transaction T whenever any of the keys T read and wrote is altered by any concurrent transaction (Marchioni 2012). In all the experiments reported in this chapter, we select as consistency criterion the latter, stronger, consistency criterion.

Detection of remote conflicts, as well as data durability, are achieved by means of a Two Phase Commit (Bernstein, Hadzilacos, & Goodman 1986) based replication protocol (2PC). To assess the performance of alternative transaction replication protocols, we developed two custom prototypes of Infinispan (ensuring the same consistency levels originally provided by Infinispan), in which we replaced the native replication protocol with two alternative protocols, i.e. Primary-Backup (PB) and a replication protocol based on Total Order Broadcast, which we refer to as TO. Note that due to the vastness of literature on transactional replication protocols, an exhaustive evaluation of all existing solutions is clearly infeasible. However, the three protocols that we consider, 2PC, PB, and TO, represent different well-known archetypal approaches, which have inspired the design of a plethora of different variants in the literature. Hence, we argue that they capture the key tradeoffs in most existing solutions. However, the protocol-agnostic approach adopted by MorphR is flexible enough to cope with other replication protocols, including, e.g., partial replication and quorum protocols. Next, for self-containment reasons, we briefly overview the three considered protocols:

2PC: Infinispan integrates a variant of the classic two phase commit based distributed locking protocol. In this scheme, transactions are executed locally in an optimistic fashion in every replica, avoiding any distributed coordination until the commit phase. At commit time, a variant of two phase commit is executed. During the first phase, updates are propagated to all replicas, but, unlike typical distributed locking schemes, locks are acquired only by a single node (called the "primary" node), whereas the remaining nodes simply acknowledge the reception of the transaction updates (without applying them). By acquiring locks on a single node, this protocol avoids distributed deadlocks, a main source of inefficiency of classic two phase commit based schemes. However, unlike the classic two phase commit protocol, the locks on the primary need to be maintained until all other nodes have acknowledged the processing of the commit. This protocol produces a large amount of network traffic, which typically leads to an increase of the commit latency (of update transactions), and suffers from a high lock duration, which can generate lock convoying at high contention levels.

PB: This is a single-master protocol allowing the processing of update transactions only on a single node, called the primary, whereas the remaining nodes are used exclusively for processing read-only transactions. The primary regulates concurrency among local update transactions using a deadlock-free commit time locking strategy, and propagates synchronously its updates to the backup nodes. Read-only transactions can be processed in a non-blocking fashion on the backups, regulated by Infinispan's native multiversion concurrency control algorithm. In this protocol, the primary is prone to become a bottleneck in write dominated workloads. On the other hand, its commit phase is simpler than in the other considered protocols (which follow a multi-master approach). This alleviates the load on the network and reduces the commit latency of update transactions. Further, by intrinsically limiting the number of concurrently active update transactions, it is less subject to trashing due to lock contention in high conflict scenarios.

TO: Similarly to 2PC, this protocol is a multi-master scheme that processes transactions without any synchronization during their execution phase. Unlike 2PC, however, the transaction serialization order is not determined by means of lock acquisition, but by relying on a Total Order Broadcast service (TOB) to establish a total order among committing transactions (Defago, Schiper, & Urban 2004). Upon their delivery by TOB, transactions are locally certified and either committed or aborted depending on the result of the certification. Being a lock-free algorithm, TO does not suffer from the lock convoying phenomena in high contention scenarios. However, its TOB-based commit phase imposes a larger communication overhead with respect to 2PC (and PB). This protocol has higher scalability potential than PB in write dominated workloads, but is also more prone to incur in high abort rates in conflict intensive scenarios.

5.1.2 Benchmarks

We consider three benchmarks representative of very heterogeneous domains, namely TPC-C (Raab 1993), Radargun[1] and Geograph (Ziparo, Cottefoglie, Calisi, Zaratti, Giannone, & Romano 2013).

TPC-C is a standard benchmark for OLTP systems, which portrays the activities of a wholesale supplier and generates mixes of read-only and update transactions with strongly skewed access patterns and heterogeneous durations. We have developed an implementation of TPC-C that was adapted to execute on a NoSQL key/value store, which include three different transaction profiles: Order Status, a read-only long running transaction; New Order, a computation intensive update transaction that generates moderate contention; and Payment, a short, conflict prone update transaction.

Radargun is a benchmarking framework designed by JBoss to test the performance of distributed, transactional key-value stores. The workloads generated by Radargun are simpler and less diverse than those of TPC-C, but they have the advantage of being very easily tunable, thus allowing to easily explore a wide range of possible workload settings.

Geograph is a benchmarking tool that allows for injecting complex and rich workloads representative of the most popular geo-social applications. Geograph has been designed to be

[1]https://github.com/radargun/radargun/wiki

Parameters Settings for the TPC-C Workloads					
	# Warehouses	% Order Status	% Payment	% New Order	# Threads
TW1	10	20	70	10	1
TW2	1	20	30	50	8
TW3	1	30	70	0	1

Parameters Settings for the Radargun Workloads						
	%Write Tx	# Reads RO Tx	# Reads Wrt Tx	# Writes (Wrt Tx)	# Keys	# Threads
RW1	50	2	1	1	5000	8
RW2	95	200	100	100	1000	8
RW3	40	50	25	25	1000	8

Parameters Settings for the Geograph Workloads				
	Agent Composition	Location	Description	# Threads
GW1	128 ReadPost10UpdatePos90 Agents	128 different sites	Normal state of the application	12
GW2	196 ReadPost10UpdatePos90 Agents and 4 Blogger Agents	Rome	Big event	12
GW3	160 ReadPost10UpdatePos90 and 10 Blogger Agents	Rome	The event has finished	12

Table 5.1: Parameters Settings

flexible both in the heterogeneity and in the dynamics of the generated workloads. In particular, it provides 19 different geo-social services (i.e. actions) and 16 application specific user simulators — called agents. Agents can be combined in complex ways to generate dynamic workload profiles.

For TPC-C we consider three different workload scenarios, which are generated by configuring the following benchmark parameters: the number of warehouses, i.e. the size of the keyset that is concurrently accessed by transactions, which has a direct impact on the generated contention; the percentage of the mix of transaction profiles generated by the benchmark; and the number of active threads at each node, which allows to capture scenarios of machines with different CPU power (by changing the number of concurrent threads active on each node). This last parameter allows to simulate, for instance, changes of the computational capacity allocated to the virtual machines hosting the data platform in a Cloud computing environment. The detailed configuration of the parameters used to generate the three TPC-C workloads, which are referred to as TW1, TW2, and TW3, are reported in Table 5.1.

For Radargun we also consider three workloads, which we denote as RW1, RW2 and RW3. These three workloads are generated synthetically, and their characteristics can be controlled by tuning three parameters: the ratio of read-only vs update transactions; the number of read/write operations executed by (read-only/update) transactions; and the cardinality of the set of keys stored in the data grid — which are accessed uniformly at random by each read/write operation. The parameters settings used for these workloads is also depicted in Table 5.1.

As for Geograph, we consider a simulation of a sports fan geo-social application where fans can trace their trips to sport events and share their opinions about it. The workload simulates an application with a few hundreds of concurrent simulated agents who submit requests with 0 think time. This allows simulating actually a much larger population of users considering that typically each user updates its position/submits requests in geo-social networks with a frequency of tens of seconds/minutes. This is the workload that one would expect from a fairly popular application with large populations of registered users. Table 5.1 reports the parameters settings used for the considered workloads. These experiments relied on two different types of agents: *bloggers*, which make small text posts (Twitter-style) without commenting, liking and searching for posts; and *ReadPost10UpdatePos90*, a type of agent that tracks its current position while reading posts in the vicinity (in this case, it reads posts 10% of the time and updates the position 90% of the time). Regarding the workloads, GW1 simulates a steady scenario in which users are spread across the world in 128 different locations. GW2 simulates the start of a major sport event in Rome, during which many users share their position, read posts about the surroundings, and comment on the event live. In GW3, the ceremony is over and the number of agents who share their position starts diminishing while more agents start blogging about the event.

5.1.3 Analysis of the results

We now report and discuss experimental data that illustrates the performance of 2PC, PB, and TO protocols under different workloads. All the experiments reported in this chapter have been obtained using a commodity cluster of 10 nodes. The machines are connected via a Gigabit switch and each one has Ubuntu 10.04 installed, 8 cores (2 Intel(R) Xeon(R) CPU E5506 @ 2.13GHz) and 32GB of RAM memory. We performed experiments with different cluster sizes. However, as the corresponding data sets show similar trends, for most workloads we only report results using 10 nodes; the only exceptions are workload TW2, for which we also depict results

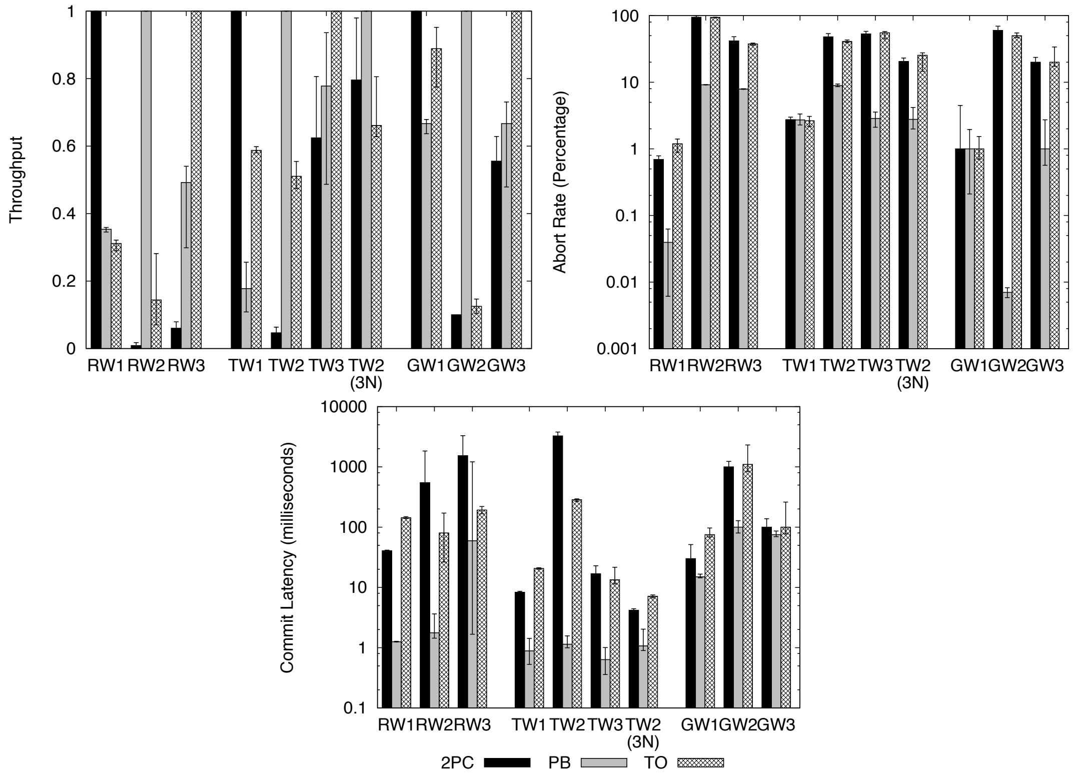


Figure 5.1: Performance of 2PC, PB and TO protocols.

in a 3 nodes cluster (denoted as TW2(3N)), and Geograph, for which we used a cluster of 5 nodes.

The first plot in Figure 5.1 reports the throughput achieved by each protocol normalized to the throughput of the best performing protocol (in the same scenario). The second and third plots report values on the transaction abort rate and commit latency (both using logscale for the y axis).

The results clearly show that none of the protocols can achieve optimal performance in all the considered workload configurations. Furthermore, the relative differences among the performance of the protocols can be remarkably high: the average normalized throughput of the worst performing protocol across all workloads is around 20% (i.e., one fifth) of the best performing protocol for each workload; also, the average throughputs across all workloads of the PB, TO, and 2PC are approximately, respectively, 30%, 40% and 50% lower than that of the optimal protocol. Furthermore, by contrasting the results obtained with TW2 using different scales of the platform, it can be observed that, even for a given fixed workload, the best performing replication protocol can be a function of the number of nodes currently in the system. These figures provide a direct measure of the potential inefficiency of a statically configured system.

The reasons underlying the shifts in the relative performance of the replication protocols can be quite intricate, as the performance of the considered protocols is affected by a number of inter-dependent factors affecting the degree of contention on both logical (i.e. data) and physical (mostly network and CPU) resources. Therefore, it may be extremely hard to manually define policies that control the adaptation. Next, we provide some insights on these factors.

In the workloads RW1, TW1 and GW1, 2PC has a good leverage over the remaining protocols. This is explainable considering that, in these low conflict configurations, 2PC does not suffer of lock convoying. This allows it to leverage on its ability to process update transactions at all nodes (unlike PB), while enjoying lower commit latencies w.r.t. TO.

The workloads RW2, TW2 and GW2, conversely, are favourable to PB in both these scenarios. In fact, the high degree of contention causes the thrashing of the two multi-master protocols (2PC and TO), which suffer from an extremely high transaction abort rate.

In the scenarios favourable to TO, namely RW3, TW3 and GW3, contention is sufficiently high to cause lock convoying on 2PC, which results in a higher commit latency. Furthermore, the workloads contain a sufficient percentage of update transactions to saturate the primary node with PB.

5.2 Architectural Overview

The architecture of MorphR is depicted in Figure 5.2. The system is composed by two macro components, a *Reconfigurable Replicated In-Memory Data Store* (RRITS), and a *Replication Protocol Selector Oracle* (RPSO).

The RRITS externalizes user-level APIs for transactional data manipulation (such as those provided by a key/value store, as in this prototype, or an STM platform), as well as APIs, used in MorphR by the RPSO, that allow its remote monitoring and control (to trigger adaptation). Internally, the RRITS supports multi-layer reconfiguration techniques, which are encapsulated by abstract interfaces allowing to plug in, in a modular fashion, arbitrary protocols for replica coherency and concurrency control. The Group Communication System (GCS) coordinates the

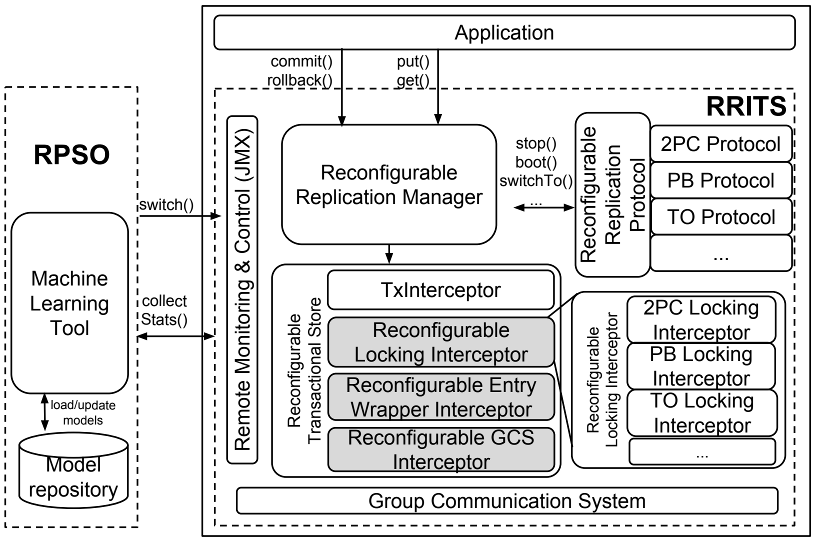

Application
commit()
rollback()
put()
get()
RRITS
RPSO
switch()
collect Stats()
Machine Learning Tool
load/update models
Model repository
Remote Monitoring & Control (JMX)
Reconfigurable Replication Manager
stop()
boot()
switchTo()
...
Reconfigurable Replication Protocol
2PC Protocol
PB Protocol
TO Protocol
...
Reconfigurable Transactional Store
TxInterceptor
Reconfigurable Locking Interceptor
Reconfigurable Entry Wrapper Interceptor
Reconfigurable GCS Interceptor
Reconfigurable Locking Interceptor
2PC Locking Interceptor
PB Locking Interceptor
TO Locking Interceptor
...
Group Communication System

Figure 5.2: MorphR's architectural overview.

communication between all nodes in the system and detects any faults that may occur. A detailed description of this building block is provided in Section 5.3.

The RPSO is an abstraction that allows encapsulating different performance forecasting methodologies. The Remote Monitoring and Control component relies on JMX to bridge interactions between RPSO and the RRITS for collecting statistics to feed the oracle and triggering a protocol switch. The oracle implementation may be centralized or distributed. In a centralized implementation, the RPSO is a single process that runs in one of the replicas or in a separate machine. In the distributed implementation, each replica has its own local instance of the oracle that coordinates with other instances to reach a common decision. In this prototype, we chose to implement the centralized version, which will be discussed in more detail in Section 5.4.

5.3 In-Memory Data Storage

The RRITS is composed by two main sub-components, the Reconfigurable Replication Manager and the Reconfigurable Transactional Store, which are described next.

5.3.1 Reconfigurable Replication Manager

The Reconfigurable Replication Manager (RRM) is the component in charge of orchestrating the reconfiguration of the replication protocol, namely the transition from a state of the system in which transactions are processed using a replication protocol A, to a state in which they are processed using a protocol B. The design of RRM was guided by the goal of achieving both generality and efficiency.

An important observation is that in order to maximize efficiency, it is often necessary to take a *white box* approach: by exploiting knowledge on the internals of the involved protocols, it is usually possible to define specialized (i.e. highly efficient) reconfiguration schemes. On the other hand, designing specialized reconfiguration algorithms for all possible pairs of protocols leads to an undesirable growth of complexity, which can hamper the platform's extensibility.

MorphR addresses this tradeoff by introducing a generic, protocol-agnostic reconfiguration protocol that guarantees the correct switching between two arbitrary replication protocols, as long as these adhere to a very simple interface (denoted as *ReconfigurableReplicationProtocol* in Figure 5.2). This interface allows MorphR to properly control their execution (stop and boot them). In order to achieve full generality, i.e. to be able to ensure consistency in presence of transitions between any two replication protocols, MorphR's generic reconfiguration protocol is based on a conservative "stop and go" approach, which enforces the termination of all transactions in the old protocol, putting the system in a quiescent state, before starting executing the new protocol.

MorphR requires that all pluggable protocols implement the methods needed by this stop and go strategy (described below), benefiting from its extensibility and guaranteeing the generality of the approach. On the other hand, in order to maximize efficiency, for each pair of replication protocols (A,B), MorphR allows for registering an additional *protocol switcher* algorithm, which interacts with the RRM via a well defined interface. The RRM also uses such specialized reconfiguration protocols, whenever available, and otherwise resorts to using the protocol-agnostic reconfiguration scheme.

Figure 5.3 depicts the state machine of the reconfiguration strategies supported by MorphR. Initially the system is in the STEADY state, running a single protocol A. When a transition to another protocol B is requested, two paths are possible. The default path (associated with the generic "stop and go" approach) first puts the system in the QUIESCENT state and then starts protocol B, which will put the system back to the STEADY state. The fast path consists of invoking the fast switching protocol. This protocol will place the system in a temporary TRANSITION state, where both protocol A and protocol B will coexist. When the switch terminates, only

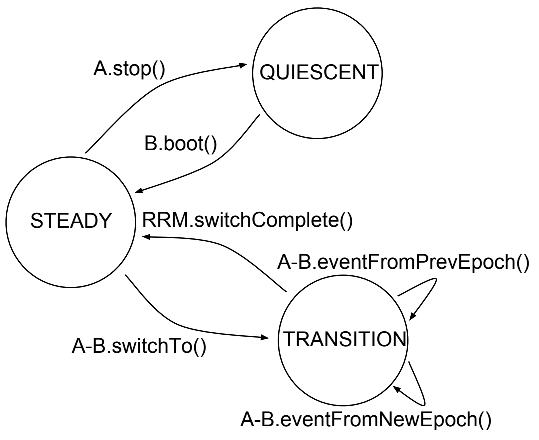


Figure 5.3: MorphR reconfiguration finite state machine.

protocol B will be executed and the system will be again in the STEADY state. For instance, the system may first switch A→B using the fast switch, and then perform the transition B→C using a stop and go switch (because it may be very hard or impossible to implement a fast switch transition B→C).

We will now discuss, in turn, each of the two protocol reconfiguration strategies supported by MorphR.

5.3.1.1 "Stop and Go" reconfiguration

The methods defined in the *ReconfigurableReplicationProtocol* interface can be grouped in two categories: i) a set of methods that allow the RRM to catch and propagate the transactional data manipulation calls issued by the application (e.g. begin, commit, abort, read and write operations), and ii) two methods, namely *boot()* and *stop()*, that every pluggable protocol must implement:

- *boot()*: This method is invoked to start the execution of a protocol from a QUIESCENT state, i.e., no transactions from any other protocol are active in the system, and implements any special initialization conditions required by the protocol.
- *stop*(*boolean* eager): This method is used to stop the execution of a protocol and putting the system in a QUIESCENT state. The protocol dependent implementation of this method must guarantee that, when it returns, there are no transactions active in the system executing with that protocol. The *eager* parameter is a boolean that allows to specify

if on-going transactions should be aborted immediately, or if the system should allow for on-going transactions to terminate before entering the QUIESCENT state.

```
stop(boolean eager) {
    block generation of new local transactions;
    if eager then
        abort any local executing transaction;
    else
        wait for completion of all local executing transactions;
    end
    broadcast (Done);
    wait received Done from all processes;
    wait for completion of all remote transactions;
}
```

Algorithm 1: *stop*() method of the 2PC protocol.

Algorithm 1 exemplifies the implementation of this interface, for the case of the 2PC. First, all new local transactions are blocked. When local transactions finish executing, a DONE message is broadcast announcing that no more transactions will be issued by this replica in the current protocol. Before returning, the *stop* method waits for this message from all the other replicas in the system and for the completion of any remote transactions executing in the that replica.

Algorithm 2 refers to the implementation of the *stop()* method for the PB protocol, distinguishing between the replica that plays the role of the primary and the remaining backup replicas. Since only the primary can process update transactions, it does not need to wait for any remote transactions to complete, so when all its local transactions finish, it broadcasts the DONE message before returning from the *stop()* method. As for the replicas, they simply wait for the DONE message from the primary before returning from the *stop()* method.

5.3.1.2 Fast switching reconfiguration

The default “stop and go” strategy ensures that, at any moment in time, no two transactions originated by different protocols can be concurrently active in the system. Non-blocking reconfiguration schemes avoid this limitation by allowing overlapping the execution of different protocols during the reconfiguration. In order to establish an order on the reconfigurations, the RRM (i.e. each of the RRM instances maintained by the nodes in the system) relies on the notion of epochs. Each fast switching reconfiguration triggers a new epoch and all transaction events (namely, prepare, commit, and abort events) are tagged with the epoch in which they were generated.

```
stop(boolean eager) {                                              /* Primary */
    block generation of new local update transactions;
    if eager then
        abort any local executing transaction;
    else
        wait for completion of all local executing transactions;
    end
    broadcast (Done);
}
stop(boolean eager) {                                              /* Backup */
    wait for Done from primary;
}
```

Algorithm 2: PB implementation of the stop() method.

To support fast switching between a given pair of protocols (oldProtocol, newProtocol), the MorphR framework requires that the programmer implements the following set of methods:

- *switchTo()*: This method is invoked to initiate fast switching. This triggers the increase of the local epoch counter on the replica, and alters the state of the replica to TRANSITION.
- *eventFromPrevEpoch*(event): This method processes an event of a transaction that was initiated in an epoch previous to the one currently active in this replica.
- *eventFromNewEpoch*(event): This method processes an event from a transaction that was initiated by a replica that is either in the TRANSITION state of the new epoch, or that has already completed the switch to new epoch and entered the STEADY state.

As also depicted by the state machine in Figure 5.3, the methods *eventFromPrevEpoch* and *eventFromNewEpoch* are only executed by a replica that has entered the TRANSITION state. Hence, whenever a replica receives either one of these two events while it is still in the STEADY state of protocol A, it simply buffers them, delaying their processing till the *switchTo()* method is invoked. As an optimization, in this case the prototype actually avoids buffering *eventFromPrevEpoch* events: this is safe because it means that the transaction's event has been generated in the same epoch as the one in which the local node is currently executing.

Further, the RRM exposes a callback interface, via the *switchComplete*() method, which allows the protocol switcher to notify the end of the transition phase to the RRM, and which causes it to transit to the STEADY state. As for the *stop()* method, a *protocol switcher* implementation must guarantee that when the *switchComplete*() method is invoked, every transaction active in the system is executing according to the final protocol. In the following paragraphs we provide two examples of fast switching algorithms, for scenarios involving protocols whose concurrent coexistence raises different types of issues, namely 2PC↔PB and 2PC↔TO.

5.3.1.2.1 Fast switch from 2PC to PB Both PB and 2PC are lock based protocols. Further, in both protocols, locks are acquired only on a designated node, called primary (both in 2PC and PB). Hence, provided that these two nodes coincide (which is the case, for instance, in this Infinispan prototype), these two specific protocols can simply coexist, and keep processing their incoming events normally. Algorithm 3 shows the pseudo-code of the fast switching for this case. As the two protocols can seamlessly execute in parallel, in order to comply with the specification of the *fast switching* interface, it is only necessary to guarantee that when the *switchComplete* callback is invoked, no transaction in the system is still executing using 2PC. To this end, when the switching is started, a LocalDone message is broadcast and the protocol moves to a transition state, causing the activation of a new epoch. In the transition state, locally generated transactions will be already processed using PB. When the LocalDone message is received from node s by some node n, it derives from the FIFO property of channels that n will not receive additional prepare messages from s. By collecting LocalDone message from each and every node in the system, each node n can attest the local termination of the previous epoch, at which point it broadcasts a RemoteDone message (line 13). The absence of transactions currently executing with 2PC across the entire system can then be ensured by collecting the latter messages from all nodes (see lines 14-15).

```
1  2PC-PB.switchTo() {
2      broadcast (LocalDone);
3  }
4  2PC-PB.eventFromPrevEpoch(event) {
5      processEvent(event, tx);
6  }
7  2PC-PB.eventFromNewEpoch(event) {
8      processEvent(event, tx);
9  }
10 upon received LocalDone from all nodes {
11     wait for completion of prepared remote 2PC txs;
12     // guarantee reconfiguration completion globally
13     broadcast (RemoteDone);
14     wait received RemoteDone from all nodes;
15     switchComplete();
16 }
```

Algorithm 3: Fast Switching from 2PC to PB.

5.3.1.2.2 Fast switch from PB to 2PC Algorithm 4 shows the pseudo-code for the fast switching between PB and 2PC. Similarly to the opposite transition, as these two protocols both acquire locks on the same node (i.e., the primary with PB and the coordinator with 2PC), we can have 2PC and PB transactions seamlessly coexist and allow replicas to enter the TRANSITION state immediately. However, in order to ensure correctness, we must guarantee that the backups enter the 2PC's STEADY state only if there are no longer active update transactions using PB. If this condition was not guaranteed, a backup could receive a commit message generated by the primary after having already entered the STEADY state of the 2PC epoch, which is prohibited by the finite state machine defining MorphR's reconfiguration strategy. In order to achieve this result, we let both the primary and the backups enter immediately the TRANSITION state (hence allowing them to process new incoming transactions using 2PC). Next, the primary waits for the completion of any locally active transaction executing using PB before broadcasting a LOCALDONE message and entering new STEADY state. The backups, on the other hand, wait for the reception of the LOCALDONE message from the primary, before finalizing the switching to 2PC.

```
PB-2PC.switchTo() {
    // enter TRANSITION state and start processing txs using 2PC
}
PB-2PC.eventFromPrevEpoch(event) {
    processEvent(event, tx);
}
PB-2PC.eventFromNewEpoch(event) {
    processEvent(event, tx);
}
upon all local update PB txs are completed {                    /* Primary */
    broadcast (LocalDone);
    switchComplete();
}
upon received LocalDone from Primary {                           /* Backup */
    switchComplete();
}
```

Algorithm 4: Fast switching from PB to 2PC.

5.3.1.2.3 Fast switch from 2PC to TO 2PC and TO protocols are radically different, as they use different concurrency control schemes (lock-based vs lock-free) and communication primitives (plain vs totally ordered broadcast) that require different data-structures/algorithms at the transactional data store level. So, it is impossible for a replica to start processing transactions with TO if any locally generated 2PC transaction is still active. To this end, the fast switch implementation from 2PC to TO, in Algorithm 5, returns from the *switchTo* method (entering the new, TO-regulated epoch) only after it has committed (or aborted) all its local transactions from the current epoch. During the TRANSITION state, a node replies negatively to any incoming prepare message for a remote 2PC transaction, thus avoiding incompatibility issues with the currently executing TO protocol. Transactions from the new TO epoch, on the other hand, can be validated (and accordingly committed or aborted). However, if they conflict with any previously prepared but not yet committed 2PC transaction, the commit of the TO transaction must be postponed until the outcome of previous 2PC transactions is known. Otherwise, it can be processed immediately according to the conventional TO protocol. Also in this case a second global synchronization phase is required to ensure the semantics of the *switchComplete*.

```
2PC-TO.switchTo() {
    block generation of new local transactions;
    wait for local transactions in prepared state;
    broadcast (LocalDone);
}
2PC-TO.eventFromPrevEpoch(event) {
    if event is of type Prepare then
        rollback(tx);
    end
    processEvent(event, tx);
}
2PC-TO.eventFromNewEpoch(event) {
    if tx conflicts with some tx' that uses 2PC then
        wait for tx' to commit or abort;
    end
    processEvent(event, tx);
}
upon received LocalDone from all nodes {
    // guarantee reconfiguration completion globally
    broadcast (RemoteDone);
    wait received RemoteDone from all nodes;
    switchComplete();
}
```

Algorithm 5: Fast switching from 2PC to TO.

5.3.1.2.4 Fast switch from TO to 2PC Algorithm 6 presents the pseudo-code for this switching. The key idea underlying this switching is to disseminate via the total order channel a special SWITCHING message, whose reception demarcates the termination of the TO epoch and the ability to start accepting transactions requesting to commit using 2PC. The usage of total order broadcast for disseminating the SWITCHING message ensures that all the replicas in the system agree on which is the last transaction to be processed according to TO. More in detail, before the reception of this message, transactions being processed using TO can be processed normally, whereas the reception of a Prepare message (generated during the first phase of 2PC) triggers the abort of the corresponding transaction. After the reception of a SWITCHING message, conversely, transactions requesting to commit using TO are deterministically aborted, whereas transactions using 2PC can be processed normally. It should be noted that this solution allows transactions that begin in the TRANSITION state to execute their logic, up to the point in which they enter their commit phase — at which point, they are aborted unless the SWITCHING message has already been received. This allows for the temporary coexistence in the system of transactions executing with both protocols. Finally, analogously to the previous fast switching protocols, before executing the *switchComplete* method, replicas execute a global synchronization phase (by exchanging LOCALDONE messages) before executing the *switchComplete* method.

```
TO-2PC.switchTo() {
    TO-broadcast (Switching);
}
TO-2PC.eventFromPrevEpoch(event) {
    if received Switching from any node then
        rollback(tx);
    else
        processEvent(event, tx);
    end
}
TO-2PC.eventFromNewEpoch(event) {
    if (not received Switching from any node ∧
      event is of type Prepare) then
        rollback(tx);
    else
        processEvent(event, tx);
    end
}
upon the completion of all local TO txs{
    broadcast (LocalDone);
}
upon received LocalDone from all nodes {
    switchComplete();
}
```

Algorithm 6: Fast switching from TO to 2PC.

5.3.1.3 Coping with failures

MorphR ensures fault tolerance as long as its three main elements are fault tolerant, namely the replication protocols, RPSO, and RRITS:

- Replication protocols must (and typically do) implement their own recovery schemes; for instance, in 2PC, if a node crashes, the protocol must be able to deal appropriately with this scenario, in order to ensure that locks are eventually released. Since the current implementation of MorphR relies on a group membership service, nodes leverage on this building block to detect and recover from failures.
- As for the RPSO, standard replication techniques, such as primary-backup or quorum-based approaches, can be used to ensure the oracle's fault-tolerance.
- Concerning the RRITS, the algorithms that govern both the "stop-and-go" and the "fast-switching" schemes were presented assuming no faults for simplicity. They contain various wait conditions that require gathering messages from all replicas; in order to avoid blocking arbitrarily on these wait conditions, a simple approach is to rely on the group membership service and wait for messages from all the nodes currently in the group.

5.3.2 Reconfigurable Transactional Store

MorphR assumes that, when a new replication protocol is activated, the *boot()* method performs all the setup required for the correct execution of that protocol. In some cases, this may involve performing some amount of reconfiguration of the underlying data store, given that the replication protocol and the concurrency control algorithms are often tightly coupled. Naturally, this setup is highly dependent of the concrete data store implementation in use.

When implementing MorphR on Infinispan, our approach to the protocol setup problem has been to extend the original Infinispan architecture in a principled way with the aim to minimize intrusiveness. To this end, we systematically encapsulated the modules of Infinispan that required reconfiguration using software *wrappers*. The wrappers intercept calls to the encapsulated module, and re-route them to the implementation associated with the current replication protocol configuration.

The architectural diagram in Figure 5.2 illustrates how this principle was applied to one of the key elements of Infinispan, namely the interceptor chain that is responsible for i) capturing commands issued by the user and by the replication protocols, and ii) redirecting them towards the modules managing specific subsystems of the data store (such as the locking system, the data container, or the group communication system). The interceptors whose behaviours had to be replaced due to an adaptation of the replication protocol, shown in gray in Figure 5.2, were replaced with generic reconfigurable interceptors, for which each replication protocol can provide its own specialized implementation. This allows to flexibly customize the behaviour of the data container depending on the replication protocol currently in use.

5.4 Selector Oracle

The Replication Protocol Selector Oracle component is a convenient form of encapsulating different performance forecasting techniques. In fact, different approaches, including analytical models (Didona, Romano, Peluso, & Quaglia 2014) and machine learning (ML) techniques (Mitchell 1997; Singh, İpek, McKee, de Supinski, Schulz, & Caruana 2007), might be adopted to identify the replication protocol on the basis of the current operating conditions. In MorphR we have opted for using ML-based forecasting techniques, as they can cope with arbitrary replication protocols, maximizing the generality and extensibility of the proposed approach, thanks to their black-box nature.

The selection of the best replication protocol lends itself naturally to be cast as a *classification* problem (Mitchell 1997), in which one is provided with a set of input variables (also called *features*) describing the current state of the system and is required to determine, as output, a value from a discrete domain (i.e., the best performing protocol among a finite set in our case). We integrated MorphR with three distinct ML-based classification techniques: the C5.0 (Quinlan a) decision tree algorithm, Neural Networks (NN) (Haykin 1994), and Support Vector Machines (SVM) (Platt 1999). C5.0 builds a decision-tree classification model during an off-line training phase in which a greedy heuristic is used to partition, at each level of the tree, the training dataset according to the input feature that maximizes information gain (Quinlan a). The output model is a decision-tree that closely classifies the training cases according to a compact (human-readable) rule-set, which can then be used to classify (i.e., decide the best performing replication strategy for) future scenarios. Regarding NN and SVM, we used the implementations available in Weka (Frank, Hall, Holmes, Kirkby, Pfahringer, & Witten 2005), a popular open-source framework that provides a common interface to a large number of ML algorithms. The NN algorithm implemented in the Weka framework trains a multi-layered network using the classic back-propagation algorithm (Rumelhart, Durbin, Golden, & Chauvin 1995) to classify instances. We used the default configuration in Weka, which generates a number of hidden layers equal to half the number of input features. Concerning the Support Vector Machine technique, we also rely on the default configuration of the Weka's SMO package, which implements John Platt's sequential minimal optimization algorithm for training a support vector classifier (Keerthi, Shevade, Bhattacharyya, & Murthy 2001). We shall discuss the methodology adopted in MorphR to build ML-based performance models shortly, and focus for the moment on discussing how these models are used at runtime.

In our current reference architecture, the RPSO is a centralized logical component, which is physically deployed on one of the replicas in the system. Although the system is designed to transparently support the placement of the RPSO on a dedicated machine, the overhead imposed to query the ML model is so limited (on the order of the microseconds), and the query frequency is so low (on the order of the minutes or of the tens of seconds), that the RPSO can be collocated on any node of the data platform without causing perceivable performance interferences.

The RPSO periodically queries each node in the system, gathering information on several metrics describing different characteristics of the current workload in terms of both contention on logical (data) and physical resources. This information is transformed into a set of input features used to query the machine learner about the most appropriate configuration. If the current protocol configuration matches the predicted one, no action is taken; otherwise a new configuration is triggered.

This approach results in an obvious tradeoff: the more often the RPSO queries the ML, the faster it reacts to changes in the workloads. However, it also increases the risk to trigger unnecessary configuration changes upon the occurrence of momentary spikes that do not reflect a real sustained change in the workload. In this prototype we use a simple approach based on a moving average over a window time of 30 seconds, which has proven successful with all the workloads we experimented with. As with any other autonomic system, in MorphR there is also a tradeoff between how fast one reacts to changes and the stability of the resulting system[2]. In the current prototype, we simple use a fixed "quarantine" period after each reconfiguration, to ensure that the results of the previous adaptation stabilise before new adaptations are evaluated. Of course, the system may be made more robust by introducing techniques to filter out outliers (Hodge & Austin 2004), detect statistically relevant shifts of system's metrics (Page 1954), or predict future workload trends (Kalman 1960). These techniques are orthogonal to the contributions of this chapter, and fall outside of its scope.

5.4.1 Construction of the ML model

The accuracy achievable by any ML technique is well known to be strictly dependent on the selection of appropriate input features (Mitchell 1997). These should be, on one hand, sufficiently rich to allow the ML algorithm to infer rules capable of closely relating fluctuations of the input variables with shifts of the target variable. On the other hand, considering an excessively high number of features leads to an exponential growth of the training phase duration and to an increase of the risk of inferring erroneous/non-general relationships (a phenomenon called overfitting (Mitchell 1997)).

After conducting an extensive set of preliminary experiments, we decided to let MorphR gather a base set of 14 metrics, namely: percentage of write transactions, number of read and write operations per read-only and write transactions and their local and total execution time, abort rate, throughput, lock waiting time and hold time, duration of the commit phase, average CPU and memory utilization.

As we will show in Section 5.5, this set of input features proved sufficient to achieve high prediction accuracy, at least for the set of replication protocols considered in this chapter. Nevertheless, to ensure the generality of the proposed approach, we allow protocol developers to enrich the set of input features for the ML by specifying, using an XML descriptor, whether the RPSO should track any additional metric that the protocol exposes via a standard JMX interface.

A key challenge to address in order to build accurate ML-based predictors of the performance of multiple replication protocols is that several of the metrics measurable at run-time can be

[2]Stability refers to the unlikelihood that a transient workload oscillation induces a spurious protocol switch.

strongly affected by the replication protocol currently in use. Let us consider the example of the transaction abort rate: in workloads characterized by high data contention, the 2PC abort rate is typically significantly higher than when using the PB protocol for the same workload, due to the presence of a higher number of concurrent (and distributed) writers. In other words, the input features associated with the same workload can be significantly different when observed from different replication protocols. Hence, unless additional information is provided that allows the ML to contextualize the information encoded in the input features, one risks feeding the ML with contradictory inputs that can end up confusing the ML inference algorithm and ultimately hinder its accuracy.

In order to address this issue, we consider three alternative strategies for building ML models: i) a simple baseline scheme, which does not provide any information to the ML concerning the currently used replication protocol; ii) an approach in which we extend the set of input features with the protocol used while gathering the features; iii) a solution based on the idea of using (training and querying) a distinct model for each different replication protocol. The second approach is based on the intuition that, by providing information concerning the "context" (i.e., the replication protocol) in which the input features are gathered, the ML algorithm may use this information to disambiguate otherwise misleading cases. The third approach aims at preventing the problem *a priori*, by avoiding the usage of information gathered using different protocols in the same model. An evaluation of these alternative strategies can be found in Section 5.5.1.

Finally, the last step of the model building phase consists in the execution of an automated feature selection algorithm, which is aimed at minimizing the risk of overfitting and maximizing the generality of the model by discarding features that are either too closely correlated among each other (and hence redundant), or too loosely correlated with the output variable (and hence useless). We rely on the Forward Selection (Guyon & Elisseeff 2003) technique, a greedy heuristic that progressively extends the set of selected features till the accuracy it achieves when using ten-fold cross-validation on the training set is maximized.

5.5 Evaluation

This section presents the results of an experimental study aimed at assessing three main aspects: the accuracy of the ML-based selection of the best-fitting replication protocol (Section

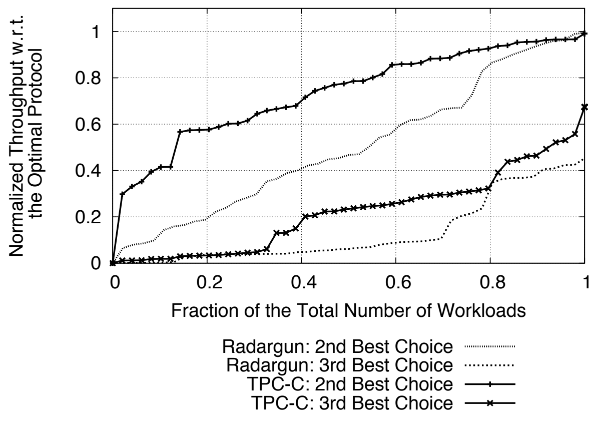


Figure 5.4: Cumulative distribution of the normalized throughput vs the optimal protocol.

5.5.1); the efficiency of the alternative protocol switching strategies discussed in Section 5.3 (Section 5.5.2); the overheads introduced by the online monitoring and re-configuration supports employed by MorphR to achieve self-tuning (Section 5.5.3).

5.5.1 Accuracy of the RPSO

In order to assess the accuracy of the RPSO with the three different ML approaches, we generated approximately 75 workloads for both TPC-C and Radargun (results for Geograph are omitted because they show similar trends), varying uniformly the parameters that control the composition of transaction mixes and their duration. In particular, for each of the 3 TPC-C workloads described in Section 5.1, we explored 25 different configurations, varying the percentages of Order Status, Payment and New Order Transactions. Analogously, starting from each of the three Radargun workloads reported in Table 5.1, we explored 27 different variations of the parameters that control the percentage of write transactions, and the number of read/write operations both in read-only and update transactions. This workload generation strategy allowed us to obtain a balanced data set containing approximately the same number of workloads for which each of the three considered protocols results to be the optimal choice.

We ran each of the above workloads with the three considered protocols, yielding a data set of approximately 1350 cases that serves as the basis for this study. Figure 5.4 provides an interesting perspective on our data set, reporting the normalized performance (i.e., committed transactions per second) of the 2nd and 3rd best choice with respect to the optimal protocol

ML	Model Type	Radargun	TPCC
C5.0	**w/o Prot.**	9.90%	17.40%
	With Prot.	3.30%	13.30%
	Three Models	3.30%	10.00%
SVM	**w/o Prot.**	14.4%	15.06%
	With Prot.	14.4%	15.52%
	Three Models	12.66%	12.32%
NN	**w/o Prot.**	5.43%	11.87%
	With Prot.	3.7%	12.32%
	Three Models	2.88%	9.13%

Table 5.2: Percentage of misclassification.

(i.e., the protocol that had the best performance) for each of the considered workloads. The plot highlights that, when considering the Radargun workloads, the selection of the correct protocol configuration has a striking effect on system's throughput: in 50% of the workloads, the selection of the 2nd best performing protocol is at least twice slower than the optimal protocol; further, the performance decreases by a factor up to 10x in 50% of the workloads in case the worst performing protocol were to be erroneously selected by the RPSO. On the other hand, the TPC-C workload shows less dramatic, albeit still significant, differences in the relative performances of the protocols. Being the performance of the three protocols relatively closer with TPC-C than with Radargun, the classification problem at hand is indeed harder in the TPC-C scenario, at least provided that one evaluates the accuracy of the ML-based classifier exclusively in terms of misclassification rate. On the other hand, in practical settings, the actual relevance of a misclassification is clearly dependent on the actual throughput loss due to the sub-optimal protocol selection. In this sense, the Radargun's workloads are significantly more challenging than TPC-C's ones. Hence, whenever possible, we will evaluate the quality of our ML-based classifiers using both metrics, i.e. misclassification rate and throughput loss vs optimal protocol.

The first goal of this evaluation is to assess the accuracy achievable by using the three alternative model building strategies described in Section 5.4, namely i) a baseline that adopts a single model built using no information on the protocol in execution when collecting the input features, ii) an approach in which we include the protocol currently in use among the input features, and iii) a solution using a distinct model per each protocol.

Table 5.2 shows the percentage of misclassification for each of the above approaches and for each of the three considered ML algorithms. These results were obtained by averaging the results

ML	**Train**	**Radargun**		**TPCC**	
		Miscl.	**Thr. Loss**	**Miscl.**	**Thr. Loss**
C5.0	**90%**	12%	1%	18%	1%
	70%	15%	2%	22%	4%
	40%	14 %	4%	25%	9%
SVM	**90%**	8%	0.3%	18%	1%
	70%	17%	7%	14%	2%
	40%	13%	2%	17%	1%
NN	**90%**	0%	0%	27%	5%
	70%	5%	1%	9%	2%
	40%	8%	0.5%	12%	2%
2nd prot.		100%	14%	100%	23%
3rd prot.		100%	51%	100%	75%
Random Choice		33%	45%	33%	35%

Table 5.3: Accuracy of the considered ML algorithms.

of ten models (for each ML algorithm), each built using ten-fold cross validation. The results show that the misclassification rate can be significantly lowered by incorporating in the model information on the protocol in use when characterizing the current workload and system's state. In particular, using distinct models for each protocol, as expected, we minimize the chances that the ML is misled by the simultaneous presence of training cases exhibiting similar values for the same feature but associated with different optimal protocols (due to being measured when running different protocols), or, vice versa, of training cases associated with the same optimal protocol but exhibiting different values for the same feature (again, because they were observed using different protocols). At the light of this result, in MorphR, as well as in the remainder of this section, we opted for using a distinct model for each protocol.

The data in Table 5.3 allows us also to compare the three considered ML methodologies from the two-fold perspective of absolute and relative accuracy in the selection of the replication protocol, by reporting the misclassification rate (i.e., the percentage of wrongly classified workloads) and the relative loss in throughput with respect to the optimal protocol. The table presents information concerning models based on training sets of different sizes, as well as for the 2nd and 3rd best performing protocol for each scenario, and for a trivial uniformly random selection strategy.

The data highlights that NN achieves globally the highest accuracy, both in absolute and relative terms, with C5.0 and SVM closely following. Interestingly, when looking at the average throughput loss due to the choice performed by the predictive models, one can notice that all the considered models achieve efficiency levels very close to the optimal one. The fact that the misclassification rate is typically significantly larger than the average throughput loss w.r.t. the optimum means that, in the cases in which the predictive models misclassify a workload, the selected protocol delivers a performance relatively close to the optimal choice. These are scenarios in which, on one hand, it is naturally harder to predict which protocol delivers the best performance, given that the two best protocols achieve very similar performance. On the other hand, precisely because of this, in these scenarios the relative gain achievable by using a "perfect" model (one that is always correct) w.r.t. either of the considered ML algorithms is typically very modest and always less than 10%. Finally, the last three rows of the table confirm that the selection of the correct replication protocol can have a strong impact on system's performance, at least for the set of workloads considered in this study. This perspective allows for better appreciating the actual effectiveness of the considered ML-based predictors. Overall, the fact that all the considered ML algorithms achieve a quite good, and relatively close, accuracy suggests that, at least considering the dataset used in this study, the problem of identifying the best performing protocol for a given workload lends itself nicely to solutions based on statistical, black-box approaches.

In order to assess the time each ML technique requires to build a model, we used the entire data set for the Radargun benchmark and ran the previously mentioned Forward Selection technique to obtain the most accurate model. C5.0 was the fastest algorithm, taking 22 seconds, while SVM and NN were 14x and 17x slower, respectively, to build the best model. When considering these figures, however, one should take into account that C5.0 is a commercial quality product implemented in C, whereas the NN and SVM implementations used in this study are coded in Java and are part of a research-oriented framework (Weka) designed to ensure extensibility and flexibility rather than maximizing performance.

In Figure 5.5 we show data highlighting the relevance of choosing the most appropriate combination of features when building ML based models. To this end the plot reports the misclassification rate achieved when using models (w/o protocol information, using C5.0 and the TPC-C benchmark) based on the several combinations of features tested during the feature selection process. The plot clearly shows the impact of the correct selection of the set of features during the model construction phase, which is in practice the most time consuming one as it typically requires generating and comparing tens of different models.

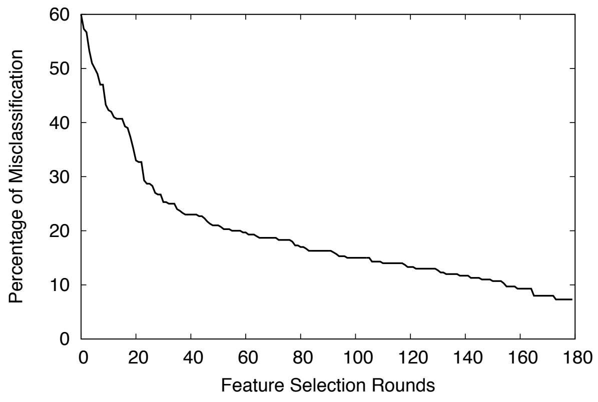
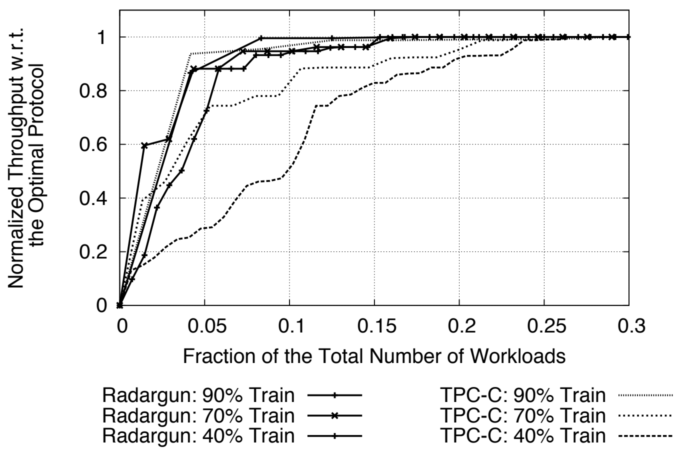


Figure 5.5: Misclassification vs feature selection.

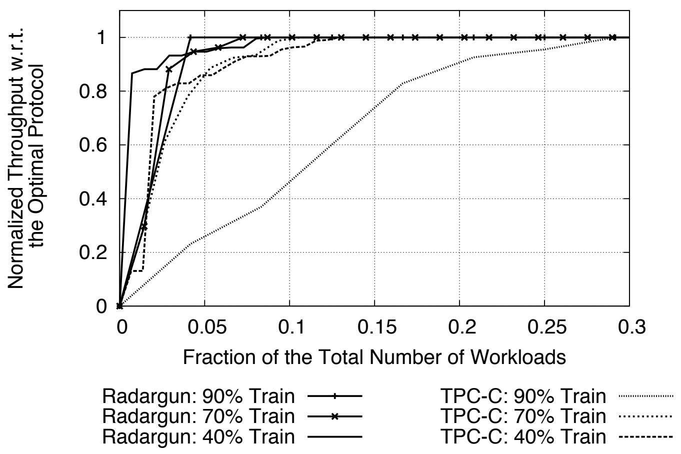
Figure 5.6 evaluates the accuracy of the RPSO from a different perspective, reporting the cumulative distribution of the throughput achievable by the RPSO's predictions for each workload, normalized to the throughput of the optimal protocol for that workload. In order to assess the extrapolation power of the classifiers built using the proposed methodology we progressively reduce the training set from 90% to 40% of the entire data set, and use the remaining cases as test sets.

Both benchmarks show the same trend for all ML approaches. As the training set becomes larger, the percentage of cases with sub-optimal throughput decreases. Furthermore, in these cases, the loss of throughput in absolute value, when compared to the optimal choice, also decreases. In fact, even for models built using the smallest training set, and considering the most challenging benchmark (namely TPC-C), the performance penalty with respect to the optimal configuration is lower than 10% in around 85% of the workloads. On the other hand, for models built using the largest training set, the throughput penalty is lower than 10% in about 90% of the cases. Again, the plots confirm that the accuracy achieved by the three ML approaches is, globally, quite similar.

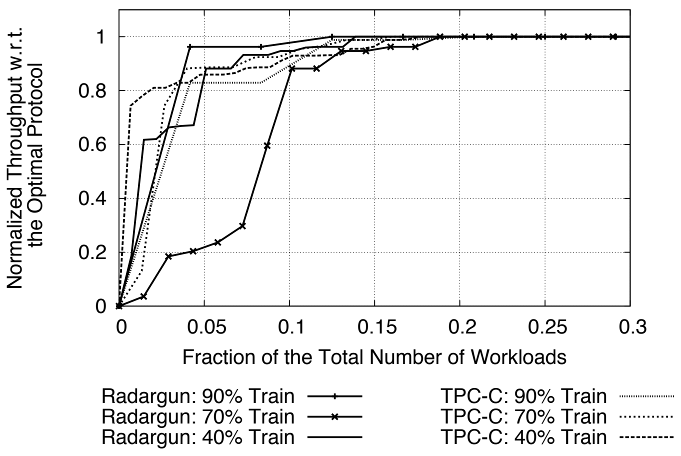
Overall, the data highlights the remarkable accuracy achievable by the proposed ML-based forecasting methodology, providing an experimental evidence of its practical viability even with complex benchmarks such as TPC-C.



(a) C5.0

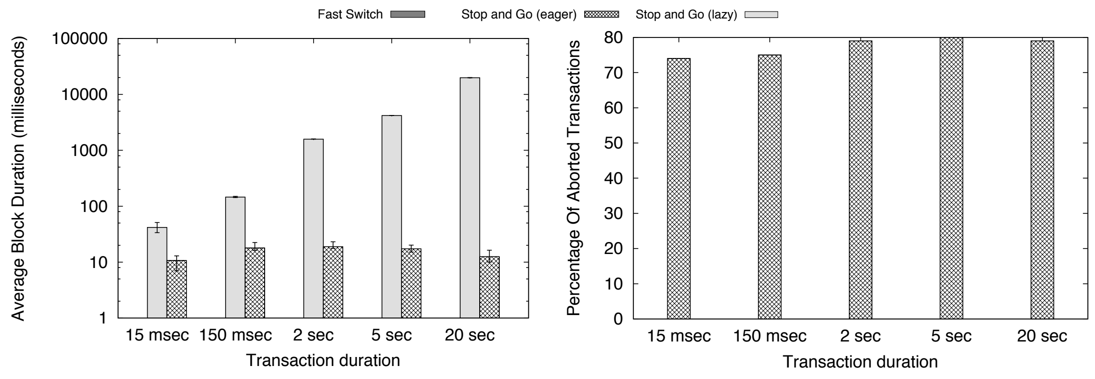


(b) NN



(c) SVM

Figure 5.6: Cumulative distribution of the normalized throughput vs the optimal protocol.



(a) Blocking time. (b) Aborts.

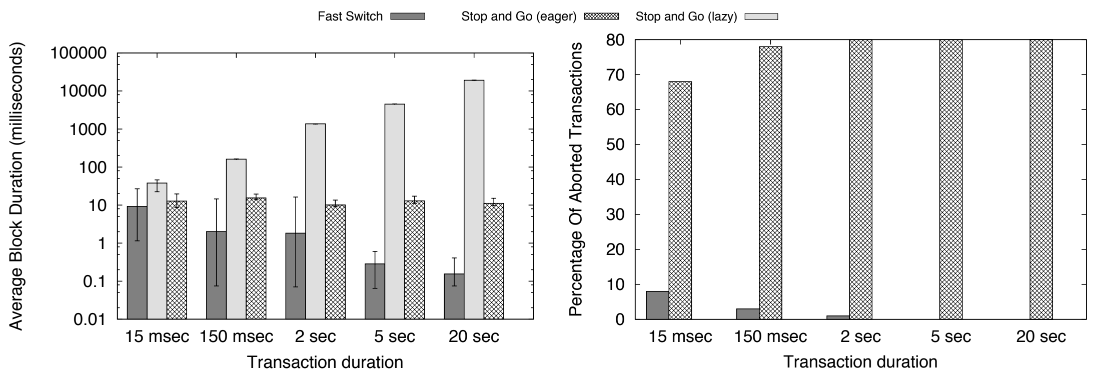
Figure 5.7: Switch between 2PC and PB.

5.5.2 Fast switch *vs* Default Switch

We now analyse the performance of specialized fast switching algorithms, contrasting it with that achievable by the generic, but less efficient, stop and go switching approach. For this purpose we built a simple synthetic benchmark designed to generate transactions with tunable duration, varying from 15 milliseconds to 20 seconds. Furthermore, we have experimented with different fast switching algorithms and both with the eager and the lazy versions of the stop and go algorithm (recall that with the eager version ongoing transactions are simply aborted, whereas with the lazy version we wait for pending transactions to terminate before switching).

We start by considering the fast switching specialized that commutes from 2PC to PB (Algorithm 3). Figure 5.7(a) shows the average blocking time, i.e., the period during which new transactions are not accepted in the system due to the switch (the shorter this period the better); the y axis is presented in logscale. Figure 5.7(b) shows the abort rate during the switching process. The figures show values for the fast-switching algorithm, and for both the lazy and eager version of stop-and-go.

The fast switching algorithm has no blocking phase and for the scenarios where the duration of transactions is larger, this can be a huge advantage when compared with the lazy stop and go approach, which in the eager version has the lowest blocking time of 10ms. As expected, the fast switching algorithm is independent of the transaction duration, as it is not required to abort or to wait for the termination of transactions started with 2PC before accepting transactions with PB. On the other hand, the lazy stop and go approach, while avoiding aborting transactions,



(a) Blocking time. (b) Aborts.

Figure 5.8: Switch between 2PC and TO.

can introduce a long blocking time (which, naturally, gets worse for scenarios where transactions have a longer duration that can go up to 10000ms). In conclusion, the eager stop and go trades a lower stopping time for a high abort rate, which can result in the abort of 80% of the transactions during switch time.

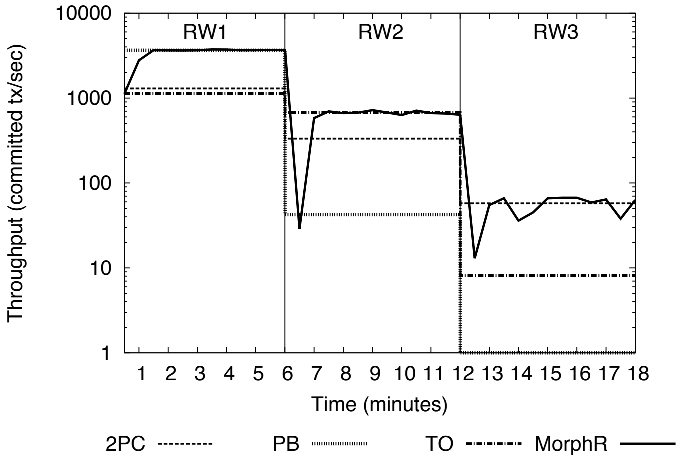
Let us now consider the fast switching algorithm for commuting from 2PC to TO (Algorithm 5), whose performance evaluation is presented in Figure 5.8. In this fast switch algorithm nodes must first wait for all local pending transactions initiated with 2PC to terminate before accepting transactions to be processed by TO. Therefore, this algorithm also introduces some amount of blocking time that, although smaller than in the case of the stop and go switching algorithm, is no longer negligible. Nevertheless, the advantages of fast switching are still very significant when transactions are very long since its blocking time does not depend of the transactions' duration (10000 ms vs. 0.1ms). The eager version of stop and go still provides a constant low blocking time (10ms), but at the cost of aborting on average 80% of the transactions during switch time.

These results show that, whenever available, the use of specialized fast switching algorithms is preferable. On the other hand, the stop and go algorithm can be implemented without any knowledge about the semantics of the replication protocols. Also, the eager version can provide reasonably small blocking times (in the order of 10ms) at the cost of aborting some transactions during the reconfiguration.

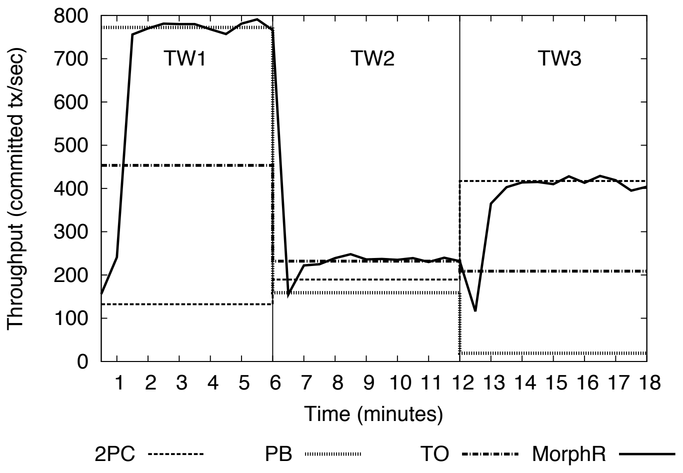
5.5.3 Performance of MorphR

Figures 5.9(a) and 5.9(b) compare the throughput of MorphR with that of a statically configured, non-adaptive version of Infinispan. In addition to the real-time performance of MorphR, we also report the average performance achieved by the three baseline protocols for all the presented workloads. The models were trained with the previously presented Radargun and TPC-C datasets, from which we removed the workloads RW1-3 and TW1-3 reported in Table 5.1. In this experiment, we injected load in the system for a duration of 6 minutes and configured the RPSO to query the system state every 30 seconds to predict the protocol to be used. The plots show that, whenever the workload changes, the RPSO detects it and promptly switches to the most appropriate protocol. As expected, the performance of MorphR keeps up with that of the best static configuration, independently of the benchmark. We can also observe that the overheads introduced by the supports for adaptivity are very reduced given that, when MorphR stabilizes, its throughput is very close to one achieved when using a static configuration.

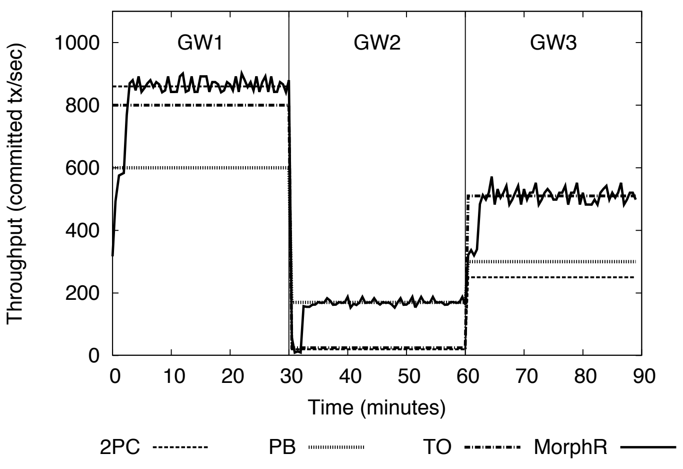
Figure 5.9(c) evaluates the effectiveness of the MorphR with Geograph. The models were trained using various Geograph workloads containing a varying percentage of agent types. The RSPO was configured to query the system every two minutes. We injected a variable workload in which the three workloads reported in Table 5.1, GW1-3, take place sequentially, each one lasting 30 minutes. We configure Infinispan to operate initially with the PB protocol, which is suboptimal given that, as shown in Figure 5.1, the optimal protocol for GW1 is 2PC. The plot highlights that the change in the workload at minute 30 leads to a drastic degradation of performance when using 2PC due to the very high data contention level of GW2. MorphR switches to PB, the optimal protocol for this workload, two minutes later. At minute 60 the workload GW3 is triggered, and we can observe an increase in the throughput due to the decrease in the workload mix of the percentage of transactions injected by Blogger agents, which are longer to process, more prone to conflict, and which generate larger commit messages than the other transaction classes. However, the optimal replication protocol for this workload is TO, which, despite generating higher contention and network load than PB, lets all nodes process update transactions. This opportunity is correctly identified by the RPSO, which triggers the switch towards TO at around minute 62, ensuring, again, the optimality of the configuration.



(a) Radargun.



(b) TPC-C.



(c) Geograph.

Figure 5.9: MorphR performance vs static configurations.

5.6 Discussion

An extensive set of works has been produced on dynamic protocol reconfiguration (Liu, van Renesse, Bickford, Kreitz, & Constable 2001; Chen, Hiltunen, & Schlichting 2001; Rütti, Wojciechowski, & Schiper 2006). A large part of this work has focused on the reconfiguration of communication protocols. For instance, the work in (Rütti, Wojciechowski, & Schiper 2006) proposes an Total Order Broadcast (TOB) generic meta-protocol that allows to stop an executing instance of a TOB protocol, and to activate a new one. This work encompasses a larger stack of software layers, and takes into account the inter-dependencies between the replica control and concurrency control schemes. Also, in MorphR we address also the issue of how to automatically determine *when* to trigger adaptation, and not only *how*.

The area of automated resource provisioning is related to this work as it aims at reacting to changes in the workload and access patterns to autonomically adapt the system's resources. Examples include works in both transactional (Didona, Romano, Peluso, & Quaglia 2014; Xiong, Chi, Zhu, Tatemura, Pu, & Hacigümü§ 2011) and non-transactional domains, such as MapReduce (Herodotou, Dong, & Babu 2011; Yigitbasi, Willke, Liao, & Epema 2013) and VM sizing (Wang, Xu, Zhao, Tu, & Fortes 2011). Analogously to MorphR, several of these systems also use machine-learning techniques to drive the adaptation. However, the problem of reconfiguring the replication protocol raises additional challenges, e.g. by demanding dedicated schemes to enforce consistency during the transitioning between two replication strategies.

To the best of our knowledge, the work in (Bhargava & Riedl 1989) pioneers the issues associated with adaptation in transactional systems (specifically, DBMSs). In particular, this chapter identifies a set of sufficient conditions for supporting non-blocking switches between concurrency control protocols. However, in order to satisfy them it is necessary to enforce very stringent assumptions (such as knowing a-priori whether the transactions will exhibit any data dependency). Our solution, on the other hand, relies on a framework that supports switching between generic replication protocols without requiring any assumption on the workload generated by applications. Several other approaches have also been proposed based on the idea to automatically analyse the incoming workload, e.g. (Martin, Elnaffar, & Wasserman 2006), to automatically identify the optimal database physical design or self-tune some of the inner management schemes, e.g. (Bruno & Chaudhuri 2007). However, none of these approaches investigated the issues related to adapt the replication scheme. Even though the work in (Ruiz-Fuertes

& Munoz-Escoi 2009) presents a meta-protocol for switching between replication schemes, it does not provide a mechanism to autonomically determine the most appropriate scheme for the current conditions.

A number of works have been aimed at automatically tuning the performance of Software Transactional Memory (STM) systems, even if most of these works do not consider replicated systems. In (Wang, Kulkarni, Cavazos, & Spear 2012), the authors present a framework for automatically tuning the performance of the system by switching between different STM algorithms. This work was based in RSTM (Spear 2010), which allows changing both STM algorithms and configuration parameters within the same algorithm. The main difference between RSTM and our work is that the latter system must stop processing transactions whenever changing the (local) concurrency control algorithm, whereas MorphR provides mechanisms allowing the coexistence of protocols while the switch is in process. The works in (Felber, Fetzer, & Riegel 2008) and (Marathe, Scherer, & Scott 2005) also allow changing configuration parameters, but our framework targets the problem of changing the protocol as a whole. Other works addressing adaptability in transaction processing include (Holanda, Brayner, & Fialho 2008; Castro, Goes, Ribeiro, Cole, Cintra, & Mehaut 2011) but they tackle different issues non-related to the reconfiguration of replication protocols.

Hybrid Transactional Replication (Kobus, Kokocinski, & Wojciechowski 2013) proposes an hybrid scheme allowing concurrent transactions to execute using the state machine replication or a deferred update approach. Contrarily to MorphR's generic approach, this hybrid replication scheme supports switching exclusively between these two specific approaches. Also, this work does not address the problem of how to determine which replication protocol to use when faced with a given workload.

MorphR is also related to the body of research on the specification of adaptation policies (Rosa, Rodrigues, Lopes, Hiltunen, & Schlichting 2013). These approaches provide an infrastructure that allows experts (such as programmers or system administrators) to control the adaptation policies of a complex system by means of different types of rules' systems. Our approach, however, relies on ML techniques to automate the determination of the adaptation policy (an idea already explored in different contexts (Didona, Romano, Peluso, & Quaglia 2014; Wang, Xu, Zhao, Tu, & Fortes 2011; Nathuji, Kansal, & Ghaffarkhah 2010)). Also, it adopts a generic software architecture that promotes extensibility/development of specialized protocols, and support arbitrary adaptations in efficient ways.

Finally, recent middleware for building distributed applications, such as Sinfonia (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007) and Chubby (Burrows 2006), also uses consensus as a building block. In the literature, a number of alternative consensus implementations have been proposed, optimized for different workloads, platform's scale etc. Therefore the ideas and techniques presented in this chapter could also be applied to these systems.

5.7 Summary

This chapter presented MorphR, a framework aimed to automatically adapt the replication protocol of STM platforms according to the current operational conditions. MorphR uses a modular approach supporting both general-purpose switching strategies, as well as optimized fast switch algorithms that can support non-blocking reconfiguration. We modelled the problem of identifying the optimal replication protocol given the current workload as a classification problem, and exploited several ML techniques to drive adaptation. MorphR has been implemented in a well-known open source transactional data grid and extensively evaluated, demonstrating its high accuracy in the identification of the optimal replication strategy and the minimal overheads introduced to support adaptability.

6 Final Remarks

Replication plays an essential role for distributed STM platforms, given that it represents the primary mean to ensure data durability. Unfortunately, no single replication technique can ensure optimal performance across a wide range of workloads and system configurations. In addition, the complexity of these systems makes configuring them manually an extremely challenging and error-prone task. This thesis addresses the problem of building a distributed STM which autonomously adapts the replication protocol in use to offer the optimal performance with varying workloads.

It began by addressing the problem of identifying the relevant features for adapting an STM system that relies on total order broadcast to ensure replica consistency. As the objective of this work was to predict the self-delivery latency of broadcast messages, several off-line machine learning techniques were tested and the results confirmed they are adequate for replicated STM generated workloads.

The following step was to design a first prototype of the autonomic replicated STM supporting a restricted family of replication protocols based on certification, named PolyCert. The proposed prototype provides adaptation on a per-transaction basis, matching each transaction with the most suited protocol, thus allowing for a finer grain adaptation. Additionally, a different machine learning technique, which only required online training, was explored, resulting in a new adaptation triggering technique to be proposed.

The third and final contribution of this thesis is an evolution of the first prototype: a framework that allows to automatically adapt the replication protocol according to the current operational conditions, named MorphR. This framework formalizes a set of interfaces with precisely defined semantics that need to be exposed (i.e. implemented) by an arbitrary replication protocol in order to support its online reconfiguration. The proposed framework is designed to ensure both generality, by means of a protocol-agnostic generic reconfiguration protocol, and efficiency, whenever the cost of the transition between two specific replication protocols can be minimized by taking into account their intrinsic characteristics. The flexibility of the proposed reconfiguration framework is demonstrated by showing that it can seamlessly encapsulate three distinct replication protocols, via both protocol-agnostic and specialized protocol switching techniques.

6.1 Open Challenges and Future Work

There are several unexplored dimensions which could complement the work presented in this thesis.

First, one of the core advantages of cloud computing is elastic scaling: computing resources can be scaled up and down easily by the cloud service provider. However, varying the number of nodes in the system impacts the behaviour of the replication protocols in DSTMs. So, the replication protocol could also be adapted to suit the new system configuration.

Also, in partially replicated systems, the number of replicas and their placement in the system impacts the number of nodes that must be contacted during a distributed transaction. This means that different replication protocols can perform differently if the replication degree or the data placement is adapted to better accommodate varying workloads, and, consequently, they should also change to a more suitable one in order to continue providing the best possible performance.

Finally, other techniques besides the black-box machine learning approach used in these works could be used to complement it, especially if the results presented in this thesis are combined with the other dimensions mentioned in the previous paragraphs. For instance, the knowledge of experts can be captured in high level policies to decide in runtime which are the most promising adaptations, from a set of possible adaptations, increasing the efficiency of the system's autonomous behaviour. One other example would be to rely on the knowledge of the developers to build analytical models to more accurately predict the behaviour of smaller components (easier to model), reducing the number of components requiring training.

References

Aguilera, M. K., A. Merchant, M. Shah, A. Veitch, & C. Karamanolis (2007, October). Sinfonia: a new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev. 41*, 159–174.

Amir, Y., C. Danilov, & J. R. Stanton (2000). A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings DSN '00*, Washington, DC, USA, pp. 327–336. IEEE Computer Society.

Andrzejak, A. & L. Silva (2008, Apr 7–11). Using machine learning for non-intrusive modeling and prediction of software aging. In *Proceedings NOMS'08*, Salvador de Bahia, Brazil.

Ansari, M., M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, & I. Watson (2011). Robust adaptation to available parallelism in transactional memory applications. In *Transactions on high-performance embedded architectures and compilers III*, pp. 236–255. Berlin, Heidelberg: Springer-Verlag.

Attiya, H., V. Gramoli, & A. Milani (2010). A provably starvation-free distributed directory protocol. In *Proceedings SSS'10*, Berlin, Heidelberg, pp. 405–419. Springer-Verlag.

Auer, P., N. Cesa-Bianchi, & P. Fischer (2002, May). Finite-time analysis of the multiarmed bandit problem. *Mach. Learn. 47*(2-3), 235–256.

Bartoli, A., C. Calabrese, M. Prica, E. Di Muro, & A. Montresor (2003). Adaptive message packing for group communication systems. In R. Meersman & Z. Tari (Eds.), *Proceedings OTM'03*, Volume 2889 of *Lecture Notes in Computer Science*, pp. 912–925. Springer Berlin Heidelberg.

Bernstein, P., D. Shipman, & W. Wong (1979). Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering 5*(3), 203–216.

Bernstein, P. A., V. Hadzilacos, & N. Goodman (1986). *Concurrency Control and Recovery in Database Systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Bershad, B., M. Zekauskas, & W. Sawdon (1993, feb). The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, pp. 528 –537.

Bhargava, B. & J. Riedl (1989, December). A model for adaptable systems for transaction processing. *IEEE TKDE 1*(4), 433–449.

Bieniusa, A. & T. Fuhrmann (2010). Consistency in hindsight: A fully decentralized stm algorithm. *Parallel and Distributed Processing Symposium, International 0*, 1–12.

Birrell, A. D. & B. J. Nelson (1984, February). Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 39–59.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*(7), 422–426.

Bocchino, R. L., V. S. Adve, & B. L. Chamberlain (2008). Software transactional memory for large scale clusters. In *Proceedings PPoPP '08*, New York, NY, USA, pp. 247–258. ACM.

Bolosky, W. J. & M. L. Scott (1993). False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Berkeley, CA, USA, pp. 3–3. USENIX Association.

Bruno, N. & S. Chaudhuri (2007). An online approach to physical design tuning. In *Proceedings ICDE'07*, pp. 826–835.

Budhiraja, N., K. Marzullo, F. B. Schneider, & S. Toueg (1993). *The primary-backup approach*, pp. 199–216. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings OSDI'06*, pp. 335–350. USENIX Association.

Cachopo, J. & A. Rito-Silva (2006, December). Versioned boxes as the basis for memory transactions. *Sci. Comput. Program. 63*, 172–185.

Carvalho, N. (2011). *A Generic and Distributed Dependable Software Transactional Memory.* Ph. D. thesis, Departamento de Engenharia Informatica, Instituto Superior Tecnico (IST), Universidade Tecnica de Lisboa.

Carvalho, N., P. Romano, & L. Rodrigues (2010). Asynchronous lease-based replication of software transactional memory. In *Proceedings Middleware '10*, Berlin, Heidelberg, pp. 376–396. Springer-Verlag.

Carvalho, N., P. Romano, & L. Rodrigues (2011a, aug.). A generic framework for replicated software transactional memories. In *Proceedings NCA' 11*, pp. 271 –274.

Carvalho, N., P. Romano, & L. Rodrigues (2011b). Scert: Speculative certification in replicated software transactional memories. In *Proceedings SYSTOR '11*, pp. 10:1–10:13. ACM.

Castro, M., L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, & J.-F. Mehaut (2011). A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings HIPC '11*, Washington, DC, USA, pp. 1–10. IEEE Computer Society.

Cecchet, E., J. Marguerite, & W. Zwaenepole (2004). C-jdbc: flexible database clustering middleware. In *Proceedings ATEC '04*, Berkeley, CA, USA, pp. 26–26. USENIX Association.

Chen, W., M. Hiltunen, & R. Schlichting (2001). Constructing adaptive software in distributed systems. In *Proceedings ICDCS'01*, pp. 635–643. IEEE Computer Society.

Coccoli, A., P. Urbán, & A. Bondavalli (2002). Performance analysis of a consensus algorithm combining stochastic activity networks and measurements. In *Proceedings DSN'02*, Washington DC, USA, pp. 551–560. IEEE Computer Society Press.

Couceiro, M. (2009, September). Cache coherence in distributed and replicated transactional memory systems. Master's thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa.

Couceiro, M., P. Romano, N. Carvalho, & L. Rodrigues (2009). D2stm: Dependable distributed software transactional memory. In *Proceedings PRDC '09*, Washington, DC, USA, pp. 307–313. IEEE Computer Society.

Couceiro, M., P. Romano, & L. Rodrigues (2011). Polycert: Polymorphic self-optimizing replication for in-memory transactional grids. In *Proceedings Middleware'11*, Lecture Notes in Computer Science, pp. 309–328. Springer Berlin / Heidelberg.

Cristian, F., R. D. Beijer, & S. Mishra (1994). A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering 1*, 177–201.

Damron, P., A. Fedorova, Y. Lev, V. Luchangco, M. Moir, & D. Nussbaum (2006). Hybrid transactional memory. In *Proceedings ASPLOS'06*, New York, NY, USA, pp. 336–346. ACM.

Dash, A. & B. Demsky (2011, aug.). Integrating caching and prefetching mechanisms in a distributed transactional memory. *Parallel and Distributed Systems, IEEE Transactions on 22*(8), 1284 –1298.

DB2 (2015). Ibm db2 database software. `http://www-01.ibm.com/software/data/db2/`.

Defago, X., A. Schiper, & P. Urban (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*(4), 372–421.

Dice, D., O. Shalev, & N. Shavit (2006). Transactional locking ii. In *Proceedings DISC'06*, Berlin, Heidelberg, pp. 194–208. Springer-Verlag.

Didona, D., D. Carnevale, S. Galeani, & P. Romano (2012). An extremum seeking algorithm for message batching in total order protocols. In *Proceedings SASO '12*, pp. 89–98. IEEE Computer Society.

Didona, D., P. Felber, D. Harmanci, P. Romano, & J. Schenker (2013). Identifying the optimal level of parallelism in transactional memory applications. *Computing Journal*, 1–21.

Didona, D., P. Romano, S. Peluso, & F. Quaglia (2014, July). Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids. *ACM Trans. Auton. Adapt. Syst. 9*(2), 11:1–11:32.

Diegues, N. & P. Romano (2014). Self-tuning intel transactional synchronization extensions. In *Proceedings ICAC'14*, pp. 209–219. USENIX Association.

Diegues, N., P. Romano, & L. Rodrigues (2014). Virtues and limitations of commodity hardware transactional memory. In *Proceedings PACT '14*, pp. 3–14. ACM.

Dietterich, T. (1995). Overfitting and undercomputing in machine learning. *ACM Comput. Surv. 27*(3), 326–327.

Ekwall, R. & A. Schiper (2007). Proceedings euro-par'07. In *Proceedings Euro-Par'07*, Volume 4641 of *Lecture Notes in Computer Science*, pp. 574–586. Springer Berlin Heidelberg.

Erman, J., A. Mahanti, M. Arlitt, I. Cohen, & C. Williamson (2007). Offline/realtime traffic classification using semi-supervised learning. *Perform. Eval. 64*(9-12), 1194–1213.

Felber, P., C. Fetzer, & T. Riegel (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings PPoPP '08*, New York, NY, USA, pp. 237–246. ACM.

Frank, E., M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, & I. H. Witten (2005). *Weka: A machine learning workbench for data mining.*, pp. 1305–1314. Berlin: Springer.

Friedman, R. & E. Hadad (2006). Adaptive batching for replicated servers. In *Proceedings SRDS '06*, pp. 311–320. IEEE Computer Society.

Garces-Erice, L. (2009). Admission control for distributed complex responsive systems. In *Proc .ISPDC '09*, Washington, DC, USA, pp. 226–233. IEEE Computer Society.

Ghodsi, A., L. O. Alima, & S. Haridi (2007). Symmetric replication for structured peer-to-peer systems. In *Proceedings DBISP2P'05/06*, Berlin, Heidelberg, pp. 74–85. Springer-Verlag.

Gramoli, V., D. Harmanci, & P. Felber (2008). Toward a theory of input acceptance for transactional memories. In *Proceedings OPODIS '08*, Berlin, Heidelberg, pp. 527–533. Springer-Verlag.

Gray, J., P. Helland, P. O'Neil, & D. Shasha (1996). The dangers of replication and a solution. In *Proceedings SIGMOD '96*, New York, NY, USA, pp. 173–182. ACM.

Gray, J. & L. Lamport (2006, March). Consensus on transaction commit. *ACM Trans. Database Syst. 31*, 133–160.

Guerraoui, R., M. Herlihy, & B. Pochon (2005). Toward a theory of transactional contention managers. In *Proceedings PODC '05*, New York, NY, USA, pp. 258–264. ACM.

Guerraoui, R. & M. Kapalka (2008). On the correctness of transactional memory. In *Proceedings PPoPP '08*, New York, NY, USA, pp. 175–184. ACM.

Guerraoui, R., M. Kapalka, & J. Vitek (2007). Stmbench7: a benchmark for software transactional memory. *SIGOPS Operating Systems Review 41*(3), 315–324.

Guerraoui, R. & L. Rodrigues (2006). *Introduction to Reliable Distributed Programming* (1st ed.). Springer Publishing Company, Incorporated.

Guyon, I. & A. Elisseeff (2003). An introduction to variable and feature selection. *J. Mach. Learn. Res. 3*, 1157–1182.

Hammond, L., V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, & K. Olukotun (2004, March). Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News 32*, 102–.

Harris, T. & K. Fraser (2003). Language support for lightweight transactions. In *Proceedings OOPSLA '03*, New York, NY, USA, pp. 388–402. ACM.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

Hendler, D., A. Naiman, S. Peluso, F. Quaglia, P. Romano, & A. Suissa (2013). Exploiting locality in lease-based replicated transactional memory via task migration. In Y. Afek (Ed.), *Distributed Computing*, Volume 8205 of *Lecture Notes in Computer Science*, pp. 121–133. Springer Berlin Heidelberg.

Herlihy, M., V. Luchangco, & M. Moir (2006). A flexible framework for implementing software transactional memory. *SIGPLAN Not. 41*(10), 253–262.

Herlihy, M., V. Luchangco, M. Moir, & W. N. Scherer, III (2003). Software transactional memory for dynamic-sized data structures. In *Proceedings PODC '03*, New York, NY, USA, pp. 92–101. ACM.

Herlihy, M. & J. E. B. Moss (1993). Transactional memory: architectural support for lock-free data structures. In *Proceedings ISCA '93*, New York, NY, USA, pp. 289–300. ACM.

Herlihy, M. & Y. Sun (2007). Distributed transactional memory for metric-space networks. *Distributed Computing 20*, 195–208.

Herodotou, H., F. Dong, & S. Babu (2011). No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings SOCC'11*, pp. 18:1–18:14. ACM.

Hodge, V. & J. Austin (2004, October). A survey of outlier detection methodologies. *Artif. Intell. Rev. 22*(2), 85–126.

Holanda, M., A. Brayner, & S. Fialho (2008). Introducing self-adaptability into transaction processing. In *Proceedings SAC'08*, pp. 992–997.

Horn, P. (2001). Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM.

Huebscher, M. C. & J. A. McCann (2008, August). A survey of autonomic computing: degrees, models, and applications. *ACM Comput. Surv. 40*(3), 7:1–7:28.

ISO (1999). *ISO/IEC 9075-2:1999: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation).* International Organization for Standardization.

Jiang, B., X. Zhang, & T. Cai (2008). Estimating the confidence interval for prediction errors of support vector machine classifiers. *J. Mach. Learn. Res. 9*, 521–540.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *J. Fluids Eng. 82*(1), 35–45.

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, & D. Lewin (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings STOC '97*, New York, NY, USA, pp. 654–663. ACM.

Keerthi, S., S. Shevade, C. Bhattacharyya, & K. Murthy (2001, March). Improvements to platt's smo algorithm for SVM classifier design. *Neural Comput. 13*(3), 637–649.

Keleher, P., A. L. Cox, S. Dwarkadas, & W. Zwaenepoel (1994). Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings WTEC'94*, Berkeley, CA, USA, pp. 10–10. USENIX Association.

Keleher, P., A. L. Cox, & W. Zwaenepoel (1992). Lazy release consistency for software distributed shared memory. In *Proceedings ISCA '92*, New York, NY, USA, pp. 13–21. ACM.

Kemme, B. & G. Alonso (1998). A suite of database replication protocols based on group communication primitives. In *Proceedings ICDCS'98*, pp. 156 –163. IEEE Computer Society.

Kemme, B. & G. Alonso (2000). Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings VLDB '00*, San Francisco, CA, USA, pp. 134–143. Morgan Kaufmann Publishers Inc.

Kemme, B., F. Pedone, G. Alonso, A. Schiper, & M. Wiesmann (2003, july-aug.). Using optimistic atomic broadcast in transaction processing systems. *Knowledge and Data Engineering, IEEE Transactions on 15*(4), 1018 – 1032.

Kephart, J. O. (2005). Research challenges of autonomic computing. In *Proceedings ICSE '05*, New York, NY, USA, pp. 15–22. ACM.

Kephart, J. O. & D. M. Chess (2003, January). The vision of autonomic computing. *Computer 36*(1), 41–50.

Kim, H., A. Raman, F. Liu, J. W. Lee, & D. I. August (2010). Scalable speculative parallelization on commodity clusters. In *Proceedings MICRO '10*, Washington, DC, USA, pp. 3–14. IEEE Computer Society.

Kobus, T., M. Kokocinski, & P. T. Wojciechowski (2013). Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings ICDCS '13*, Washington, DC, USA, pp. 286–296. IEEE Computer Society.

Kotselidis, C., M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, & I. Watson (2008, sept.). Distm:

A software transactional memory framework for clusters. In *Proceedings ICPP '08*, pp. 51 –58.

Kotselidis, C., M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, & I. Watson (2010, april). Clustering jvms with software transactional memory support. In *Proceedings IPDPS'10*, pp. 1 –12.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput. 28*(9), 690–691.

Lamport, L. (1998, May). The part-time parliament. *ACM Trans. Comput. Syst. 16*, 133–169.

Lev, Y., M. Moir, & D. Nussbaum (2007, aug). PhTM: Phased transactional memory. In *Proceedings TRANSACT '07.*

Li, K. & P. Hudak (1986). Memory coherence in shared virtual memory systems. In *Proceedings PODC'86*, pp. 229–239. ACM.

Li, K. & P. Hudak (1989). Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst. 7*(4), 321–359.

Liu, X., R. van Renesse, M. Bickford, C. Kreitz, & R. Constable (2001). Protocol switching: Exploiting meta-properties. In *ProceedingsWARGC'01*, Washington, DC, USA, pp. 37–42. IEEE Computer Society.

Maldonado, W., P. Marlier, P. Felber, J. Lawall, G. Muller, & E. Riviere (2011). Deadline-aware scheduling for software transactional memory. In *Proceedings DSN '11*, Los Alamitos, CA, USA, pp. 257–268. IEEE Computer Society.

Manassiev, K., M. Mihailescu, & C. Amza (2006). Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings PPoPP '06*, New York, NY, USA, pp. 198–208. ACM.

Marathe, V., W. Scherer, & M. Scott (2005). Adaptive software transactional memory. In *Proceedings DISC'05*, Volume 3724 of *Lecture Notes in Computer Science*, pp. 354–368. Springer Berlin / Heidelberg.

Marathe, V. J. & M. L. Scott (2004, Jun). A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept.

Marchioni, F. (2012). *Infinispan Data Grid Platform.* Packt Publishing, Limited.

Martin, M., C. Blundell, & E. Lewis (2006). Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters 5*(2), 17.

Martin, P., S. Elnaffar, & T. Wasserman (2006). Workload models for autonomic database management systems. In *Proceedings ICAS'06*, Washington, DC, USA, pp. 10. IEEE Computer Society.

Minh, C. C., M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, & K. Olukotun (2007). An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings ISCA '07*, New York, NY, USA, pp. 69–80. ACM.

Miranda, H., A. Pinto, & L. Rodrigues (2001, apr). Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings ICDCS '01*, pp. 707 –710.

Mirza, M., J. Sommers, P. Barford, & X. Zhu (2007). A machine learning approach to tcp throughput prediction. In *Proceedings SIGMETRICS '07*, New York, NY, USA, pp. 97–108. ACM.

Mitchell, T. (1997). *Machine learning*. McGraw Hill series in computer science. McGraw-Hill.

Moir, M. (1997). Transparent support for wait-free transactions. In *Distributed Algorithms*, Volume 1320 of *Lecture Notes in Computer Science*, pp. 305–319. Springer Berlin / Heidelberg.

Nathuji, R., A. Kansal, & A. Ghaffarkhah (2010). Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings Eurosys'10*, pp. 237–250. ACM.

Nitzberg, B. & V. Lo (1991, aug). Distributed shared memory: a survey of issues and algorithms. *Computer 24*(8), 52 –60.

Oracle (2015). Oracle database. `http://www.oracle.com/us/products/database/index.html`.

Page, E. (1954). Continuous inspection schemes. *Biometrika*, 100–115.

Paiva, J. a., P. Ruivo, P. Romano, & L. Rodrigues (2014, December). Autoplacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems 9*(4), 19:1–19:30.

Palmieri, R., F. Quaglia, & P. Romano (2010, july). Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Proceedings NCA'10*, pp. 20 –27.

Patiño Martínez, M., R. Jiménez-Peris, K. Bettina, & G. Alonso (2000). Scalable replication in database clusters. In *Proceedings DISC '00*, London, UK, UK, pp. 315–329. Springer-Verlag.

Payer, M. & T. R. Gross (2011). Performance evaluation of adaptivity in software transactional memory. In *Proceedings ISPASS '11*, Washington, DC, USA, pp. 165–174. IEEE Computer Society.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *Philosophical Magazine 2*(6), 559–572.

Pedone, F., R. Guerraoui, & A. Schiper (2003). The database state machine approach. *Distributed and Parallel Databases 14*, 71–98.

Perelman, D., A. Byshevsky, O. Litmanovich, & I. Keidar (2011). Smv: Selective multi-versioning stm. In *Proceedings DISC'11*, pp. 125–140.

Platt, J. (1999). Advances in kernel methods. Chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization, pp. 185–208. MIT Press.

Powell, D. (1996, April). Group communication. *Commun. ACM 39*, 50–53.

Protic, J., M. Tomasevic, & V. Milutinovic (1996, summer). Distributed shared memory: concepts and systems. *Parallel Distributed Technology: Systems Applications, IEEE 4*(2), 63 –71.

Quinlan, J. C5.0/see5.0. `http://www.rulequest.com/see5-info.html`.

Quinlan, J. R. Cubist. `http://www.rulequest.com/cubist-info.html`.

Quinlan, J. R. (1992). Learning with continuous classes. In *Proceedings AI'92*, Singapore, pp. 343–348. World Scientific.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Raab, F. (1993). TPC-C-the standard benchmark for online transaction processing (OLTP).

Ramadan, H. E., C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, & E. Witchel (2007). Metatm/txlinux: transactional memory for an operating system. In *Proceedings ISCA '07*, New York, NY, USA, pp. 92–103. ACM.

Raz, Y. (1995). The dynamic two phase commitment (d2pc) protocol. In *Proceedings ICDT '95*, London, UK, pp. 162–176. Springer-Verlag.

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society 58*(5), 527–35.

Romano, P., N. Carvalho, & L. Rodrigues (2008, September). Towards distributed software transactional memory systems. In *Proceedings LADIS'08*, Watson Research Labs, Yorktown Heights (NY), USA. (invited paper).

Romano, P. & M. Leonetti (2012, Jan). Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *Proceedings ICNC'12*, pp. 786–792. IEEE Computer Society.

Romano, P., L. Rodrigues, N. Carvalho, & J. Cachopo (2010, April). Cloud-tm: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev. 44*(2), 1–6.

Rosa, L., L. Rodrigues, A. Lopes, M. Hiltunen, & R. Schlichting (2013, March). Self-management of adaptable component-based applications. *IEEE Trans. Softw. Eng. 39*(3), 403–421.

Rughetti, D., P. Di Sanzo, B. Ciciani, & F. Quaglia (2012). Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proceedings MASCOTS'12*. IEEE.

Ruivo, P., M. Couceiro, P. Romano, & L. Rodrigues (2011, dec). Exploiting total order multicast in weakly consistent transactional caches. In *Proceedings PRDC'11*, Pasadena, California, USA.

Ruiz-Fuertes, M. & F. Munoz-Escoi (2009). Performance evaluation of a metaprotocol for database replication adaptability. In *Proceedings SRDS'09*, pp. 32–38. IEEE Computer Society.

Rumelhart, D. E., R. Durbin, R. Golden, & Y. Chauvin (1995). Backpropagation: The basic theory. In *Backpropagation*, pp. 1–34. Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Rütti, O., P. Wojciechowski, & A. Schiper (2006). Structural and algorithmic issues of dynamic protocol update. In *Proceedings IPDPS'06*, pp. 133–133. IEEE Computer Society.

Saad, M. M. & B. Ravindran (2011a). Hyflow: a high performance distributed software transactional memory framework. In *Proceedings HPDC '11*, New York, NY, USA, pp. 265–266. ACM.

Saad, M. M. & B. Ravindran (2011b). Snake: control flow distributed software transactional memory. In *Proceedings SSS'11*, Berlin, Heidelberg, pp. 238–252. Springer-Verlag.

Saballus, B., J. Eickhold, & T. Fuhrmann (2008, August 25 – 31,). Global accessible objects (gaos) in the ambicomp distributed java virtual machine. In *Proceedings SENSORCOMM'08*, Cap Esterel, France. IEEE Computer Society.

Scherer III, W. N. & M. L. Scott (2004, Jul). Contention management in dynamic software transactional memory. In *Proceedings Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada.

Schintke, F., A. Reinefeld, S. Haridi, & T. Schütt (2010). Enhanced paxos commit for transactions on dhts. In *Proceedings CCGRID '10*, Washington, DC, USA, pp. 448–454. IEEE Computer Society.

Schiper, N., P. Sutra, & F. Pedone (2010). P-Store: Genuine partial replication in wide area networks. In *Proceedings SRDS' 10*, Washington, DC, USA, pp. 214–224. IEEE Computer Society.

Schneider, F. B. (1993). *Replication management using the state-machine approach*, pp. 169–197. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Schutt, T., F. Schintke, & A. Reinefeld (2006). Structured overlay without consistent hashing: Empirical results. In *Proceedings CCGRID '06*, Washington, DC, USA, pp. 8–. IEEE Computer Society.

Schütt, T., F. Schintke, & A. Reinefeld (2008). Scalaris: reliable transactional p2p key/value store. In *Proceedings ERLANG '08*, New York, NY, USA, pp. 41–48. ACM.

Shavit, N. & D. Touitou (1997). Software transactional memory. *Distributed Computing 10*, 99–116.

Shevade, S. K., S. S. Keerthi, C. Bhattacharyya, & K. R. K. Murthy (2000). Improvements to the smo algorithm for svm regression. *IEEE Trans. Neural Netw. Learning Syst. 11*(5), 1188–1193.

Shpeisman, T., A.-R. Adl-Tabatabai, R. Geva, Y. Ni, & A. Welc (2009). Towards transactional memory semantics for c++. In *Proceedings SPAA '09*, New York, NY, USA, pp. 49–58. ACM.

Shrivastava, S., G. Dixon, & G. Parrington (1991, jan). An overview of the arjuna distributed programming system. *Software, IEEE 8*(1), 66 –73.

Singh, K., E. İpek, S. McKee, B. R. de Supinski, M. Schulz, & R. Caruana (2007, December). Predicting parallel application performance via machine learning approaches: Research articles. *Concurr. Comput. : Pract. Exper. 19*(17), 2219–2235.

Spear, M. F. (2010). Lightweight, robust adaptivity for software transactional memory. In *Proceedings SPAA '10*, New York, NY, USA, pp. 273–283. ACM.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001, August). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev. 31*, 149–160.

Usui, T., R. Behrends, J. Evans, & Y. Smaragdakis (2010, October). Adaptive locks: Combining transactions and locks for efficient concurrency. *J. Parallel Distrib. Comput. 70*(10), 1009–1023.

Vetter, J. S. & D. A. Reed (1999). Managing performance analysis with dynamic statistical projection pursuit. In *Proceedings Supercomputing '99*, New York, NY, USA, pp. 44. ACM.

Wang, A., M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, & M. Michael (2012). Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings PACT'12*, pp. 127–136. ACM.

Wang, L., J. Xu, M. Zhao, Y. Tu, & J. Fortes (2011). Fuzzy modeling based resource management for virtualized database systems. In *Proceedings MASCOTS'11*, pp. 32–42. IEEE Computer Society.

Wang, Q., S. Kulkarni, J. Cavazos, & M. Spear (2012, January). A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim. 8*(4), 54:1–54:23.

Wolpert, D. H. (1996). The lack of a priori distinctions between learning algorithms. *Neural Comput. 8*(7), 1341–1390.

Xiong, P., Y. Chi, S. Zhu, J. Tatemura, C. Pu, & H. HacigümüŞ (2011). Activesla: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings SOCC'11*, pp. 15:1–15:14.

Xu, J., M. Zhao, J. Fortes, R. Carpenter, & M. Yousif (2008). Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing 11*(3), 213–227.

Yang, L., J. M. Schopf, C. L. Dumitrescu, & I. Foster (2006). Statistical data reduction for efficient application performance monitoring. In *Proceedings CCGRID '06*, Washington, DC, USA, pp. 327–334. IEEE Computer Society.

Yigitbasi, N., T. L. Willke, G. Liao, & D. Epema (2013). Towards machine learning-based auto-tuning of mapreduce. In *Proceedings MASCOTS'13*, pp. 11–20. IEEE Computer Society.

Yoo, R. M., C. J. Hughes, K. Lai, & R. Rajwar (2013). Performance evaluation of intel&reg; transactional synchronization extensions for high-performance computing. In *Proceedings SC'13*, pp. 19:1–19:11. ACM.

Yoo, R. M. & H.-H. S. Lee (2008). Adaptive transaction scheduling for transactional memory systems. In *Proceedings SPAA '08*, New York, NY, USA, pp. 169–178. ACM.

Zhang, B. & B. Ravindran (2009). Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *Proceedings OPODIS '09*, Berlin, Heidelberg, pp. 48–53. Springer-Verlag.

Zhang, B. & B. Ravindran (2011). A quorum-based replication framework for distributed software transactional memory. In *Proceedings OPODIS '11*, Berlin, Heidelberg. Springer-Verlag.

Zhang, S., I. Cohen, J. Symons, & A. Fox (2005). Ensembles of models for automated diagnosis of system performance problems. In *Proceedings DSN '05*, Washington, DC, USA, pp. 644–653. IEEE Computer Society.

Ziparo, V., F. Cottefoglie, D. Calisi, M. Zaratti, F. Giannone, & P. Romano (2013). D4.3 - prototype of pilot application i. In *Cloud-TM project.*