Cloud-TM: an elastic, self-tuning transactional store for the cloud

João Barreto
Instituto Superior Técnico – Technical University Lisbon /INESC-ID

Pierangelo Di Sanzo
Sapienza Università di Roma

Roberto Palmieri
Sapienza Università di Roma

Paolo Romano
Instituto Superior Técnico - Technical University Lisbon /INESC-ID

ABSTRACT

By shifting data and computation away from local servers towards very large scale, world-wide spread data centers, Cloud Computing promises very compelling benefits for both cloud consumers and cloud service providers: freeing corporations from large IT capital investments via usage-based pricing schemes, drastically lowering barriers to entry and capital costs; leveraging the economies of scale for both services providers and users of the cloud; facilitating deployment of services; attaining unprecedented scalability levels.
However, the promise of infinite scalability catalyzing much of the recent hype about Cloud Computing is still menaced by one major pitfall: the lack of programming paradigms and abstractions capable of bringing the power of parallel programming into the hands of ordinary programmers.
This chapter describes Cloud-TM, a self-optimizing middleware platform aimed at simplifying the development and administration of applications deployed on large scale Cloud Computing infrastructures.

INTRODUCTION

The rapidly expanding market of commercial Cloud Computing infrastructures currently offers solutions of different flavors. Depending on the nature of the resources made available on demand by the Cloud platform, such flavors include Infrastructure-As-A-Service (IAAS), Platform-As-A-Service (PAAS) and Software-As-A-Service (SAAS). While some of these solutions are reminiscent of the application service provider (ASP) paradigm, in practice, cloud computing platforms work differently than ASPs. Examples include those offered by Amazon Web Services, AT&T's Synaptic Hosting, AppNexus, GoGrid, Rackspace Cloud Hosting, and to an extent, the HP/Yahoo/Intel Cloud Computing Testbed, and the IBM/Google cloud initiative.
Instead of owning, installing, and maintaining the software for their costumers (often in a multi-tenancy architecture), cloud computing vendors typically maintain little more than the hardware, and give customers a set of virtual machines in which to install their own software. However, getting additional computational resources is not as simple as a magic upgrade to a bigger, more powerful machine on the fly (with commensurate increases in CPUs, memory, and local storage); rather, the additional resources are typically obtained by allocating additional server instances to a task. For example, Amazon's Elastic Compute Cloud (EC2) apportions computing resources in small, large, and extra large virtual private

server instances, the largest of which contains no more than eight cores. If an application is unable to take advantage of the additional server instances by offloading some of its required work to the new instances which run in parallel with the old instances, then having the additional server instances available will not be much help.

Thus, one of the main challenges that needs to be faced to bring about the potential of cloud computing, and ultimately consolidate its business model, is the development of programming models and tools that simplify the design and implementation of applications for the cloud, so as to bring the power of parallel computing into the hands of ordinary programmers.

Unfortunately, designing and implementing software services that are actually able to match the scalability potentialities of large scale, shared-nothing Cloud infrastructures is far from being a trivial task.

One of the most crucial issues to tackle when developing large scale distributed application is certainly related to how to manage concurrent manipulations to the shared state of the application. The challenge here is to identify mechanisms that able to ensure adequate consistency levels while being:

1. simple and familiar for the programmers;
   highly efficient and scalable;
2. fault-tolerant and highly available.

Decades of literature and field experience in areas such as replicated databases, Web infrastructures, and high performance computing have led to the development of a plethora of different approaches to ensure state consistency in distributed platforms, and taught a fundamental, general lesson. The design space of distributed state consistency mechanisms is so vast that no universal, one-size-fits-all solution exists, as the efficiency of individual state management approaches is strongly affected by both:

1. the characteristics of the incoming workload, such as the ratio of read/write operations, as well as the spatial/temporal locality in the data access patterns, and
2. the scale of the system, e.g. number of nodes and local vs. geographical distribution.

The complexity of this problem is hence further exacerbated in Cloud Computing platforms precisely because of the feature that is regarded as one of the key advantages of the cloud: its ability to elastically acquire or release resources, de facto dynamically varying the scale of the platform in real-time to meet the demands of varying workloads.

This chapter describes the architecture of a novel middleware platform for service implementation in Cloud Computing platforms that is being developed in the context of the EU project Cloud-TM.

At the core of the Cloud-TM platform lies the abstraction of a Distributed Software Transactional Memory (DSTM). DSTM is a recently proposed extension of the Transactional Memory (TM) programming paradigm, which was originally introduced to simplify the development of concurrent, though not distributed, programs.

TMs free programmers from the pitfalls of conventional explicit lock-based thread synchronization by relying on the proven, familiar notion of atomic transactions to transparently guarantee the consistency of concurrent memory manipulations by multithreaded applications. By relishing the programmer from the burden of managing locks or other error-prone low-level concurrency control mechanisms, TMs have been shown to enable a significant boost in productivity, shortening development times, and increasing code reliability in complex concurrent applications.

DSTMs enrich the traditional TM model, to breach the boundaries of a single machine and transparently leverage the resources of commodity, shared-nothing clusters, hiding the complexities underlying the implementation of consistent, scalable and fault-tolerant data distribution and replication schemes. DSTMs are a recent research topic, which represents, in some sense, the confluence of the research areas on TM, distributed shared memory (DSM) and database replication. The few currently available DSTMs (Cachopo, 2007; Kotselidis et al., 2008; Bocchino, Adve, & Chamberlain, 2008; Manassiev, Mihailescu, & Amza, 2006; 20; Carvalho, Romano, & Rodrigues, 2010) have shown very promising preliminary results, highlighting how the reliance on the atomic transaction abstraction allows devising highly

efficient synchronization schemes that avoid the well-known performance limitations of classical DSM systems while ensuring strong consistency guarantees and scaling up to hundreds of nodes in data center environments.
On the other hand, existing DSTM solutions are all preliminary prototypes that have been experimented only on a restricted number of workloads. Further, existing DSTM platforms lack essential support for cloud computing platforms, such as transparent data caching, automatic data partitioning, fault-tolerance and persistence, delegating to the programmers the responsibility of implementing these low-level, error prone mechanisms.
Building on the abstraction of DSTM, the Cloud-TM project aims at an innovative data platform for cloud environments that addresses the following major challenges:

- offering a simple and intuitive programming model for the implementation of services in Cloud computing platforms that will allow a major reduction in the costs of the development process by letting service developers focus on delivering differentiating business value instead of managing low-level, error prone mechanisms such as inter-process synchronization, caching, persistence and fault tolerance;
- maximizing the scalability and efficiency (i.e. the costs/benefits ratio in the Cloud Computing usage-based pricing model) of the user-level services via autonomic mechanisms allowing to transparently alter the state consistency schemes adopted at the middleware level in face of workload fluctuations and on the basis of the amount of resources currently acquired in the cloud;
- minimizing the monitoring and administration costs by automatizing the provisioning of resources from the Cloud Computing infrastructure, based on user specified target QoS/operational costs criteria.

It has to be noted that, at the time of writing, the Cloud-TM project is still ongoing. Consequently, the development of several of the components of the Cloud-TM platform is still ongoing, and component integration is still at an early stage. The focus of this chapter will therefore be on illustrating the key motivations and challenges that were accounted for during the design phase of the Cloud-TM platform, as well as on illustrating some of the results achieved so far by the researchers involved in the project.

The remainder of the chapter is organized as follows. We start by introducing the background to Cloud-TM, describing its most relevant related work. We then present an overview of the Cloud-TM platform, addressing its main high-level components. Finally, we discuss future research directions and draw conclusions.

BACKGROUND

Programming paradigms for the Cloud. The promise of infinite scalability catalyzed much of the recent hype about Cloud Computing. However, this promise is still menaced by one major pitfall: the lack of programming paradigms and abstractions capable of bringing the power of parallel programming into the hands of ordinary programmers. This is a hot research area and several novel programming paradigms for simplifying large scale computations across shared nothing clusters have been recently proposed.
MapReduce (Jeffrey & Sanjay, 2008) represents probably one of the first, and more popular programming paradigms explicitly targeted to meet the scalability challenges of large-scale cloud infrastructures. MapReduce is a functional programming model which permits automatic parallelization and execution on large scale clusters. By forcing developers to adhere to a restricted, though neatly defined, programming model, the MapReduce's run-time system is able to automatically take care of issues such as input data partitioning, scheduling the program's execution across multiple machines and handling of node failures. MapReduce is being extensively used in large scale Google data centers to analyze in parallel huge data sets in domains such as web log and graph analysis. Its automatic parallelization and fault-tolerance features have attracted the attention of a growing community of enthusiastic users that have developed a complete open source porting of the original proprietary system, Hadoop (Hadoop Wiki, 2012).

Nevertheless, it is nowadays widely recognized that, depending on the nature of the problem to be addressed, casting a known solution algorithm into the functional MapReduce programming model might be far from being trivial, possibly forcing to fragment the computation into a sequence of MapReduce tasks or inducing unnatural additional steps which can lead to significant performance drawbacks with respect to conventional parallel programming approaches (Ranger, Raghuraman, Penmetsa, Bradski, & Kozyrakis, 2007).
To address this issue a number of MapReduce extensions, e.g. Cascading, DryadLINQ (Yu, 2008), or Pig (Olston, Reed, Srivastava, Kumar, & Tomkins, 2008), have been recently proposed. These solutions greatly simplify the approach to large data analysis problems, not forcing developers to ”think” in MapReduce, but rather exposing simpler SQL-like programming interfaces, which are then automatically mapped to an underlying MapReduce implementation.
However, the actual efficiency of MapReduce is currently matter of a controversial debate (Abadi, 2009; DeWitt & Stonebraker, 2009), with several well-known scientists critically highlighting several performance issues of MapReduce and its derivatives, based both on a comparison with the mechanisms supported by modern parallel databases, as well as on benchmarking results showing MapReduce to be about an order of magnitude slower than alternative systems (Hadoop Wiki, 2012).

Transactional Memory. Along the route of simplify the exploitation of advantages offered by Cloud Computing, data manipulation is of course another major hot topic. In particular, the techniques typically used for managing the concurrency among data and taking care of their availability and reliability need to be revised in order to exploit the Cloud Computing advantages. Moreover, the recent adoption of affordable multicore and manycore chips as the architecture-of-choice for mainstream computing is demanding a radical shift in the way software is developed, moving parallel programming from the niche of scientific and high performance computing to ordinary application domains. Unfortunately, writing scalable parallel programs using traditional lock-based synchronization primitives is a hard, time consuming, and error-prone task, mastered by only a minority of programmers. A first issue with lock-based synchronization is associated with the choice of the lock granularity. Coarse grained locking strategies, e.g. based on a single, mutual exclusion section, are of course very simple to program, but can seriously hamper parallelism. On the other hand, the design of fine grained locking schemes is an extremely challenging engineering task. First because ensuring the correctness of a lock-based program is very difficult, as a single misplaced or missing lock may easily compromise the consistency of data. Second, it is difficult to compose applications that rely on locks without knowing their internals.
Transactional Memories (TMs) respond exactly to the urge for a simpler programming model for concurrent, multi-threaded applications. When using TMs the programmers are simply required to specify which operations on shared data structures are to be executed within the scope of an atomic and isolated transaction. The task of ensuring the consistent execution of the transactions is delegated to the TM, which transparently takes care of regulating the concurrent access to shared data, by automatically aborting unserializable transactions, avoiding deadlocks and priority inversions.
Originally introduced in the seminal paper from Herlihy and Moss (Herlihy, Eliot, & Moss, 1993), as a purely hardware implemented mechanism, the first software realization of the same idea, referred to as Software Transactional Memory (STM), was proposed by Shavit and Touitou (1995). Since these seminal papers, the research on Transactional Memory remained mostly dormant until 2003, when the interest in the area was spurred again 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 deserved much attention in computer science and engineering, with hundreds of papers published in prestigious international conferences and journals addressing a wide range of complementary aspects including hardware and operating systems (OSs) support (Ramadan et al., 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). In the last years several STMs have been released (e.g., (Moir, 1997; Herlihy et al., 2003; Felber, Fetzer, & Riegel, 2008;

Cachopo, & Rito-Silva, 2006; Carvalho, Cachopo, Rodrigues, & Rito-Silva, 2008; Ramadan et al., 2007)).

An important conclusion that has been highlighted by the large body of research on STMs, and in particular by some recent, independent results (Guerraoui, Herlihy, & Pochon, 2005; Riegel, Fetzer, & Felber, 2008; Sonmez, Cristal, Harris, Unsal, & Valero, 2009), is that the search for a "one size fits all" solution in the vast multi-dimensional design space of STMs remains inconclusive: up to date, no single "panacea" solution has been identified that is able to maximize the performances of any STM workload.

Even if STMs have garnered a huge research interest over the last years, most of the research efforts in this area have been in the context of non-distributed, cache coherent, shared-memory systems. By contrast, the problem of how to extend the STM abstraction across the boundaries of a single machine and to employ transactions as a first class abstraction for large scale distributed programming has only very recently started to be addressed. However this topic needs to be considered mandatory if the facilities offered by STMs for managing transactions want to be brought in Cloud Computing environments in which, typically, much more then few computational nodes are involved. Indeed, only a handful of distributed STM (DSTM) platforms (Couceiro, Romano, Carvalho, & Rodrigues, 2009; Aguilera, Merchant, Shah, Veitch, & Karamanolis, 2007; Manassiev, Mihailescu, & Amza, 2006; Kotselidis et al., 2008; Bocchino et al., 2008) have been proposed.

Distributed Transactional Memory. Two decades of research have clearly highlighted that distributed shared memories (DSMs) are capable of achieving good performance and scalability only provided that programmers embrace relaxed consistency models (Keleher, Cox, & Zwaenepoel, 1992). Unfortunately, relaxed consistency models are typically challenging for ordinary programmers, because they are forced to understand all the subtleties of complicated consistency properties to avoid endangering correctness, and this contrast with the goal of simplifying the development of concurrent application to the programmers. On the other hand, as highlighted by the experimental evaluation of several of the aforementioned DSTM platforms (Manassiev, Mihailescu, & Amza, 2006; Kotselidis et al., 2008; Bocchino et al., 2008), DSTMs use the transaction abstraction not only as way to simplify parallel programming but also as a natural means to aggregate communication efficiently, avoiding the performance pitfalls proper of DSM systems without sacrificing programming simplicity. In fact, unlike strongly consistent DSMs, which require expensive remote synchronizations at each single memory access, atomic transactions allow for optimistic implementations that permit to batch any consistency action within a single synchronization phase taking place at commit time (Aguilera et al., 2007; Kotselidis et al., 2008; Couceiro et al., 2009). This approach amortizes communication overheads across a (possibly large) number of memory accesses, with clear benefits in terms of performance. Taking a closer look at existing DSTM platforms, it is relatively easy to draw a line between solutions, such as those in (Kotselidis et al., 2008; Manassiev, Mihailescu, & Amza, 2006; Couceiro et al., 2009), which were designed for being deployed in small scale clusters, and those, such as Cluster-STM (Bocchino et al., 2008) and Sinfonia (Aguilera et al., 2007), which were architected to scale up to several hundreds of nodes.

In the former group of solutions, all of them are based on full replication schemes and, when fault-tolerance is addressed, this is done by using Group Communication Systems that provide support for Virtual Synchrony and Atomic Broadcast (Amir, Danilov, & Stanton, 2000; Miranda, Pinto, & Rodrigues, 2001). These strategies have shown encouraging performance results when employed in clusters of at most 10 nodes, but it is very unlikely that they sustain the scalability challenges of the largest cloud computing environments.

Sinfonia (Aguilera et al., 2007) and Cluster-TM (Bocchino et al., 2008), on the other hand, have shown that DSTM platforms can achieve impressive scale-ups (up to several hundreds of nodes) thanks to the usage of lightweight and highly optimized distributed commit protocols and partial replication schemes.

Unfortunately, none of these solutions have been designed for operating in a Cloud Computing environment, characterized by an elastic, dynamic scaling of the number of nodes in the DSTM platform. This raises a number of important technical issues related both to the efficiency and consistency of the

DSTM platform. On the performance side, the efficiency of key mechanisms used to ensure the consistency of the DSTM is strongly affected by the number of replicas and by the cost of inter-replica communication (which may be highly heterogeneous and variable in Cloud platforms, such as the Amazon EC2, made up of large datacenters spread across the globe). For instance, distributed contention management approaches that rely on a centralized coordinator, or fully replicated caching schemes, are known to be a winning solution in small-scale clusters (Kotselidis et al., 2008; Couceiro et al., 2009). However, centralized contention management schemes are also known to suffer of scalability problems, as the coordinating node is naturally prone to become the system's bottleneck. Despite their higher complexity, data partitioning and partial replication are definitely more appealing in large scale data centers than full replication.

Automated Resource Provisioning. Another highly desirable feature for a DSTM platform operating in a Cloud Computing platform is the ability to automatically determine how many resources should be acquired from the cloud to face the current load pressure. However, existing approaches to the problem of automatic resource provisioning in Cloud Computing, e.g. (Kalyvianaki, Charalambous, & Hand, 2009; Xu, Zhao, Fortes, Carpenter, & Yousif, 2007), cannot be straightforwardly used in the context of a DSTM platform. As just discussed, in fact, the performance of DSTM platforms are deeply affected by a number of factors (e.g. commit/caching schemes, workload characteristics) that are peculiar to the DSTM approach and require novel ad hoc cost/performance forecasting models. Elastically scaling up and down the number of nodes of a DSTM platform also requires facing consistency issues, such as the release of a node holding the freshest copy of a data item, e.g. in multi-master replication schemes (Pacitti, Coulon, Valduriez, & Özsu, 2005), cannot obviously determine data losses.

Relational databases and Cloud-oriented storage. In traditional, multi-tiered service-oriented applications persistence support is typically provided by a relational database, which also ensures ACID transactional guarantees. Unfortunately, the reliance on relational databases to synchronize concurrent accesses to application data represents a largely suboptimal solution for what concerns both the ease of programming and the efficiency/scalability of the system. For what concerns programming simplicity, an important issue is related to the mismatch between the object oriented model that is used to code the application logic and the relational model supported by commercial off-the-shelf DBMSs. Since the mapping between the relational model and the object-oriented model is far from being trivial, it is common practice to use platforms (Sun MicroSystems, 2003) or generic frameworks (Hibernate, 2009; iBATIS, 2009) that perform or facilitate the mapping between the relational database and the object-oriented model. Unfortunately, these approaches represent a patch to the problem, rather than a solution to it, and can bias the development of an object-oriented rich domain model (Cachopo, & Rito-Silva, 2006) and be a source of overheads and complexity.
Generally speaking, current support for persistence in cloud computing environments focuses on high availability of resources, on-demand scalability, flexibility of the persistent representation, and on simplifying the distribution of the resources through the cloud. On the other side, they do not focus yet on being programmer-friendly. Instead, they force application design to meet certain API limitations. In some aspects, this is a step back in some of the best object-oriented programming practices. They also impose an impedance mismatch and the programmer must explicitly code the persistency of the non-transient data, implying a cost in development time. Last but not least, existing persistency mechanisms either cannot be composed with transactions, or just support the persistency of data local to a node in the context of a transaction.

OVERVIEW OF THE CLOUD-TM PLATFORM

Overview

Figure 1. Architectural Overview of the Cloud-TM Platform
(CloudTmArchitectureOverview.tif)

The high level architecture diagram of the Cloud-TM platform is presented in Figure 1. This diagram will be used as a starting point to derive, in a top-down fashion, more refined versions of the platform's main building blocks, which will be described in detail in the remainder of this document.
The Data Platform is responsible for storing, retrieving and manipulating data across a dynamic set of distributed nodes, elastically acquired from the underlying IaaS Cloud provider(s). It will expose a set of APIs, denoted as "Data Platform Programming APIs" in Figure 1, aimed at increasing the productivity of Cloud programmers from a twofold perspective:

1. To allow ordinary programmers to store and query data into/from the Data Platform using the familiar and convenient abstractions provided by the object-oriented paradigm, such as inheritance, polymorphism, associations.
2. To allow ordinary programmers to take full advantage of the processing power of the Cloud-TM platform via a set of abstractions that will hide the complexity associated with parallel/distributed programming, such as load balancing, thread synchronization and scheduling, fault-tolerance.

Lower in the stack we find the backbone of the Data Platform, namely a highly scalable, elastic and dynamically Reconfigurable Distributed Software Transactional Memory (RDSTM).
Cloud-TM uses Red Hat's Infinispan (JBoss/Red Hat, 2001) as the base component that implements the RDSTM. Infinispan is a recent in-memory transactional data grid designed from the ground up to be extremely scalable. Cloud-TM extends Infinispan with innovative algorithms (in particular for what concerns data replication and distribution aspects), and real-time self-tuning schemes aimed at guaranteeing optimal performance even in highly dynamic Cloud environments.
At its lowest level, the Data Platform supports the possibility to persist its state over a wide range of heterogeneous durable storage systems, ranging from local/distributed file systems to Cloud storages (such as Amazon's S3 or Cassandra).
The Autonomic Manager is the component in charge of automating the elastic scaling of the Data Platform, as well as of orchestrating the self-optimizing strategies that will dynamically reconfigure the data distribution and replication mechanisms to maximize efficiency in scenarios entailing dynamic workloads.
Its topmost layer will expose an API allowing the specification and negotiation of QoS requirements and budget constraints.
The Autonomic Manager leverages on pervasive monitoring mechanisms that do not only track the utilization of heterogeneous system-level resources (such as CPU, memory, network and disk), but also characterize the workload sustained by the various subcomponents of the transactional Data Platform (local concurrency control, data replication and distribution mechanisms, data contention level) and their efficiency.
The stream of raw data gathered by the Workload and Performance Monitor component is then filtered and aggregated by the Workload Analyzer, which distils workload profiling information and alert signals that serve as input for the Adaptation Manger.
Finally, the Adaptation Manager hosts a set of optimizers that rely on techniques of different nature, including analytical or simulation-based models and machine-learning-based mechanisms. These optimizers self-tune the various components of the Data Platform and control the dynamic auto-scaling mechanism. Their ultimate goal is to meet QoS/cost constraints.

The next subsections describe each main high-level components of Cloud-TM in greater detail.

Data Platform

This section is devoted to discuss the architectural aspects associated with the development of the key building blocks of the Cloud-TM Data Platform. In particular we will describe the APIs that will be exposed by the Data Platform to the programmers, and the architectural organization of the main building blocks that will expose these APIs.

As depicted in Figure 1, the APIs exposed by the Data Platform can be grouped into three main building blocks:

• **The Object Grid Mapper.** A key feature of the Cloud-TM platform is the support for the development of object-oriented programs based on object domain models; that is, programs that maintain their states as sets of entities, which are represented by instances of various classes with relationships among them.

As we will discuss more in detail later on the chapter, the Transactional In-Memory Data Grid component of the Cloud-TM architecture is a key-value store and its API is not the most adequate for a programmer that wants to store a large graph of entities interconnected via complex relationships. So, the proposed architecture of the Cloud-TM platform includes a layer on top of the Transactional In-Memory Data Grid that is responsible for providing the higher-level API needed to develop an application that is based on an object-oriented domain model.

• **The Search API**. Any complex data-centric application requires supporting ad-hoc queries to retrieve and manipulate portions of the state that it manages. Given that we want the Data Platform to provide support for development of object-oriented applications, the module in charge of implementing the querying functionality should be able to deal with some intrinsic aspects of the object-oriented model, e.g. supporting notions such as polymorphism and inheritance.

These functionalities will be implemented by the Search API component. This component will expose to the programmer the Java Persistent API - Query Language (JPA-QL) interface, which represents, at the time of writing, the industry standard for encoding queries on an object-oriented database at least for what concerns the Java platform. This same API will be used to support advanced full-text queries, supporting notions such as ranked searching, search by fields, proximity queries, phrase queries etc.

Under the hood, this component will rely on an innovative design strategy that will integrate some of the leading open-source projects in the area of data management and indexing, namely Hibernate OGM and Lucene, with a fully-fledged distributed query engine providing support for complex data manipulation/queries (such as joins or aggregate queries).

• **The Distributed Execution Framework.** This framework will provide a set of APIs aimed at simplifying the development of parallel applications running on top of the Cloud-TM platform. It will essentially consist of two main parts:

1. An adaptation of the java.util.concurrency framework, providing a set of building blocks for the development of classic imperative parallel applications deployed on the Cloud-TM platform. These will include abstractions such as task executors and synchronizers (e.g. countdown latches) and transaction-friendly concurrent data collections.
2. An adapted version of the popular Google's MapReduce framework. This will allow developers to transparently parallelize their tasks and execute them on a large cluster of machines, consuming data from the underlying in-memory data grids rather than using input files as it was defined by the original proposal.

We address each of these modules in the following subsections.

Object grid mapper

The Object Grid Mapper module is responsible for implementing a mapping from an object-oriented domain model to the Infinispan's key-value model. Thus, a programmer that is developing an application to execute in the Cloud-TM platform will use the API provided by this module not only to develop his application's object-oriented domain model, but also to control the life-cycle of the application's entities.

Given the maturity of the Java Persistence API (JPA) (Oracle, 2012) and its wide acceptance by Java software developers, we decided that the Cloud-TM platform should provide an implementation of the

JPA standard as one of its options for mapping an object-oriented domain model. The adoption of JPA will make the Cloud-TM platform standards compliant, easing its adoption by the masses of programmers already familiar with JPA, and providing a smoother migration path for those applications already built on top of JPA. Hibernate OGM, discussed below, is the implementation of JPA for the Cloud-TM platform.
Cloud-TM also offers another API based on the Domain Modeling Language (DML) of the Fénix Framework (Cachopo, 2007). The main reason for having an alternative is because JPA imposes some constraints on the implementation of the mapping from objects to the underlying distributed transactional memory platform that may make it difficult or completely prevent some of the approaches that we would like to explore in the Cloud-TM project.

Search APIs

The Search APIs provides two main functionalities, namely full-text search queries and support for the JPA query language (JP-QL).

Full text queries. The engine supporting full text queries in the Cloud-TM platform is based on two popular open source frameworks, Hibernate Search and Apache Lucene. These have been extended in order to manipulate/query/index data maintained in the Cloud-TM's Distributed Transactional Memory platform.
Hibernate Search is an Object/Index Mapper: it consists of an indexing and an index search component. Both are backed by Apache Lucene, which is a (non-object oriented) full-text indexing and query engine. As a result of this composition, Hibernate Search allows performing full-text queries over object-oriented databases. In the following, it is briefly described how these two components interact, and how they are being integrated in the Cloud-TM platform.
Hibernate Search uses the Lucene index to search an entity and return a list of managed entities. Hence, it saves the programmer the tedious task of taking care of the object to Lucene document mapping. Further, each time an entity is inserted, updated or removed in/from the database, Hibernate Search keeps track of this event (through the Hibernate Core event system) and schedules an update of the Lucene index, in a totally transparent fashion for the programmer. This makes the data in the datastore inlined with the index information and thus makes queries accurate.
To interact with Apache Lucene indexes, Hibernate Search has the notion of DirectoryProviders. A directory provider will manage a given Lucene Directory type. Traditional directory types are file system directories and (non-distributed) n-memory directories.
Integration with the Cloud-TM's data platform is being pursued by developing a Lucene directory capable of efficiently persisting the index information that will be stored in a distributed fashion, in the Cloud-TM's in-memory transactional data grid. This way, the Cloud-TM's data store will contain both the data and the meta-data associated to it.
To maximize efficiency, Hibernate Search batches the write interactions with the Lucene index. There are currently two types of batching. Outside a transaction, the index update operation is executed right after the actual database operation. This is really a no-batching setup. In the case of an ongoing transaction, the index update operation is scheduled for the transaction commit phase and discarded in case of transaction rollback. The batching scope is therefore the transaction. This provides two immediate benefits:

- Performance: By amortizing indexing cost across multiple update operations analyzed in a batch, the Lucene indexing can achieve much higher throughput levels.
- ACIDity: The work executed has the same scoping as the one executed by the database transaction and is executed if and only if the transaction is committed. This is not ACID in the strict sense of it, but ACID behavior is rarely useful for full text search indexes since they can be rebuilt from the source at any time.

JP-QL support. Support for JP-QL APIs in the Cloud-TM platform is being achieved in two steps:

1. convert JP-QL queries into Lucene queries, and then execute them via the Hibernate Search APIs;
2. convert JP-QL queries into Teiid queries where Teiid sources will be the various Lucene indexes exposed via Hibernate Search.

As a first step, we are building a JP-QL parser that converts JP-QL queries into one or more Lucene queries. This approach is limited in what JP-QL queries are supported, as it can only provide for simple selective queries (e.g. where user.age > 30, where order.price > 10 and user.city = 'Paris') and *-to-one joins.

In order to support more complex queries, such as aggregate or join operations, the next step will consist of integrating Teiid within the Cloud-TM platform. Teiid (JBoss/Red Hat, 2012b) is a data virtualization system that allows applications to use data from multiple, heterogeneous data stores. The heart of Teiid is a high-performance query engine that processes relational, XML, XQuery and procedural queries from federated data sources. Teiid's features include support for homogeneous schemas, heterogeneous schemas, transactions, and user defined functions.

Using Teiid, the query execution would:

- convert the JP-QL query into a Teiid query tree;
- let Teiid convert the query tree into one or more Hibernate Search / Lucene queries;
- execute Lucene queries on each dedicated index;
- aggregate and or do the Cartesian join if necessary;
- return the results to Hibernate OGM.

Distributed Execution Framework

The Distributed Execution Framework (DEF) aims at providing a set of abstractions to simplify the development of parallel applications. This will allow ordinary programmers to take full advantage of the processing power available by the set of distributed nodes of the Cloud-TM platform without having to deal with low level issues such as load balancing, thread synchronization and scheduling, fault-tolerance, or asymmetric processing speeds in different nodes.

The DEF will consist of two main parts:

1. An extension of the java.util.concurrency framework, designed to transparently support, on top of the in-memory transactional data grid:
   - execution of Java's Callable tasks, which may consume data from the underlying data grid;
   - synchronization of tasks/threads via, e.g., queues, semaphores, countdown latches and other classic concurrency tools;
   - concurrent, transactional data collections, such as sets, hashmaps, skiplists, arrays;
   - processing of data streams, via the definition of pipelines of data stream processing tasks.
2. An adapted version of the popular Google's MapReduce (Jeffrey & Sanjay, 2008) framework. MapReduce is a programming model and a framework for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate k. MapReduce framework enables users to transparently parallelize their tasks and execute them on a large cluster of machines. The framework is defined as "adapted" because the input data for map reduce tasks is taken from the underlying in-memory data grid rather than using input files as it was defined by Google's original proposal.

Unlike most other distributed frameworks, the Cloud-TM DEF uses data from the transactional data platform's nodes as input for execution tasks. This provides a number of relevant advantages:

- The availability of the transactional abstraction for the atomic, thread-safe manipulation of data allows drastically simplifying the development of higher level abstractions such as concurrent data collections or synchronization primitives.
- The DEF capitalizes on the fact that input data in the transactional data grid is already load-balanced (using a Consistent Hashing scheme (Karger, 1997)). Since input data is already balanced, the

execution of tasks will be automatically balanced as well; users do not have to explicitly assign tasks to specific platform's nodes. However, our framework accommodates users to specify arbitrary subsets of the platform's data as input for distributed execution tasks.

- The mechanisms used to ensure the fault-tolerance of the data platform, such as redistributing data across the platform's nodes upon failures/leaves of nodes, can be exploited to achieve failover of uncompleted tasks. In fact, when a node F fails, the data platform's failover mechanism will migrate, along with the data that used to belong to F, also any task T that was actively executing on F.
- Both node failover and task failover policy will be pluggable. Our initial implementation will define interfaces to implement various node failover policies. Currently, we provide only a simple policy that throws an exception if a node fails. In terms of task failure the default initial implementation will simply re-spawn the failed task until it reaches some failure threshold. Future implementations might migrate such a task to another node, etc.
- As the in-memory transactional data grid will provide extensive support for data replication, more than one node can contain the same entry. Running the same task on the same node would lead to redundant computations, useful possibly for fault-tolerance reasons, but otherwise superfluous. In order to avoid this problem, it is necessary to schedule the processing of disjoint input data in nodes, keeping into account the fact that they possibly maintain replicas of the same data. Thus, the input entries for the task have to be split by the framework behind the scene. Once split, approximately the same number of non-overlapped input entries will be fed to the task on each replica. Ideally, the split input entries do not need to be merged or re-split. However, there are two cases that need to be considered. The task might run faster on a certain replica while slower on some others, even if the input entries were split evenly depending on how the task was implemented by the user. In such a case, the idle nodes could process the input entries that were not processed yet by the busy nodes (i.e. work stealing) only if the cost of work-stealing does not cancel the gain. Also, during task execution, a replica can go offline for some reason. In such a case, other replicas have to take over the input entries that the offline replica was assigned to. This is basically implemented in the same way with work stealing because we can consider the offline node as "the busiest one". Note that, once the input entries are fed to a node, they can split again to fully utilize all CPU cores. Proper scheduling needs to be done so that only the same number of threads with the number of available cores run at the same time, in order to avoid excessive context switching. Work stealing also needs to be implemented to address the problem mentioned above in a local level.

Distributed Execution Model. The main interfaces for distributed task execution are DistributedCallable and DistributedExecutorService. DistributedCallable is a subtype of the existing Callable from the java.util.concurrent package. DistributedCallable can be executed in a remote JVM and receive input from the transactional in-memory data grid. The task's main algorithm could essentially remain unchanged, only the input source is changed. Existing Callable implementations will most likely get their input in the form of some Java object/primitive while DistributedCallable gets its input from the underlying transactional data platform in form of key/value pairs. Therefore, programmers who have already implemented the Callable interface to describe their task units would simply extend their implementation to match DistributedCallable and use keys from the data grid as input for the task. Implementation of DistributedCallable can in fact continue to support implementation of an already existing Callable while simultaneously be ready for distributed execution by extending DistributedCallable.
DistributedExecutorService is a simple extension of the familiar ExecutorService interface from the java.util.concurrent package. Existing Callable tasks, instead of being executed in JDK's ExecutorService, are also eligible for execution on the distributed Cloud-TM data platform. The DEF would migrate a task to one or more execution nodes, run the task and return the result(s) to the calling node.

Of course, not all Callable tasks will benefit from parallel distributed execution. Excellent candidates are long running and computationally intensive tasks that can run concurrently and/or tasks using input data that can be processed concurrently.
The second advantage of the DistributedExecutorService is that it allows a quick and simple implementation of tasks that take input from Infinispan cache nodes, execute certain computation and return results to the caller. Users would specify which keys to use as input for specified DistributedCallable and submit that callable for execution on the Cloud-TM platform. The DEF's runtime would locate the appropriate keys, migrate DistributedCallable to target execution nodes and finally return a list of results for each executed Callable. Of course, users can omit specifying input keys, in which case Infinispan would execute DistributedCallable on all keys for a specified data store.

Map Reduce Execution Model. The MapReduce model supported by the Cloud-TM platform is an adaptation of Google's original MapReduce. There are four main components in each map reduce task: Mapper, Reducer, Collator and MapReduceTask.
Each implementation of a Mapper interface (see Listing 4) is a component of a MapReduceTask that is invoked once for each input entry <K,V>, namely a key/value pair of the underlying in-memory data grid. Given a cache entry <K,V> input pair, every Mapper instance migrated to a node of the data platform, transforms that input pair into intermediate key/value pair emitted into a provided Collator. Intermediate results are further reduced using a Reducer.
Finally, MapReduceTask is a distributed task uniting Mapper, Reducer and Collator into a cohesive large scale computation to be transparently parallelized across the Cloud-TM Data Platform nodes. Users of MapReduceTask need to provide a cache whose data is used as input for this task. The DEF's execution environment will instantiate and migrate instances of provided mappers and reducers seamlessly across Infinispan nodes.

Reconfigurable Software Transactional Memory

The implementation of the in-memory transactional data grid is based on Infinispan (JBoss/Red Hat, 2001), a recent open source project led by JBoss/Red Hat. In the remainder of this section we will first provide a high level overview of the current architecture of Infinispan, and then focus on how it will be enriched to meet the Cloud-TM's requirements of dynamic reconfiguration.

Infinispan's main operational modes

Infinispan architecture is extremely flexible and supports a range of operational modes.

Standalone (non-clustered) mode: in this mode, Infinispan acts as a Software Transactional Memory. Unlike classical STMs, however, Infinispan does not ensure serializability (or opacity) (Guerraoui & Kapałka, 2008), but, in order to maximize performance, it guarantees more relaxed consistency models. Specifically, it ensures the ISO/ANSI SQL isolation levels Read-committed and Repeatable-read isolation levels (JBoss/Red Hat, 2001), in addition to what is called Write skew anomaly avoidance. In the following, we briefly overview these isolation criteria, and how they are ensured.

- In the read committed isolation level, whenever the application wants to read an entry from Infinispan for the first time, the application is provided directly with the current value in the underlying data container that only contains committed values. When and if the application writes an entry, the modifications are not immediately applied; conversely, the updated data value is stored in a private transaction's workspace. Any later read by the same transaction returns the value that had previously written.
- In the repeatable read isolation level when the application issues for the first time a read request for an entry, it is provided with the value from the committed values container; this value is stored in a private transaction's workspace, and is returned whenever a new read operation is issued on that same data item.

- If the write-skew check is enabled, Infinispan checks if, between a read and a write operation on a data item X issued from a transaction A, there has been some transaction B that has updated (and committed) X. In this case the transaction A is aborted.

Independently from the isolation level, read operations are always lock-free; on the other side, every write operation requires the acquisition of a lock in order to avoid concurrent modifications on the same entry. Infinispan provides the per-entry lock mode, meaning that each <key,value> pair is protected by its own lock, and the striped lock mode, meaning that each lock is shared by a set of <key,value> pairs. This second option reduces memory consumption for lock management, but has the drawback of incurring in false lock contentions. Lock contention is solved through a simple and configurable timeout scheme; further, Infinispan provides a simple and lightweight deadlock detection mechanism to detect cyclic dependencies between pairs of transactions.

Full replication mode: in this mode, Infinispan is deployed over a set of interconnected nodes, each one holding a replica of the same key set. This mode is mainly intended for small scale deployments, given that the need to propagate updates to all the nodes in the system hampers the global scalability of this operational mode. The replication mechanism employed by Infinispan is based on the usage of the classical two-phase commit protocol (Gray & Reuter, 1993), which ensures that, in order for a transaction to be committed, it must acquire locks successfully across all the nodes in the platform (thus ensuring replica-wide agreement on the transaction serialization order).

More in detail, upon the issuing of a commit request from a transaction, the prepare phase is initiated: the set of the keys updated by the transaction together with the relevant modification that it wants to perform is broadcast to all nodes. The transaction is then re-run remotely on every node following the aforementioned locking scheme; if a node can successfully acquire all the locks needed, it sends an acknowledgement to the issuer of the prepare phase, otherwise it communicates its impossibility to proceed.

If all nodes send a positive reply, then the node of the original transaction sends a final commit message and all nodes can safely propagate the modifications on the affected keys.

Partial replication mode: in this mode every Infinispan replica stores a subset of the data globally managed by the system; every <key, value> pair is replicated (using a Consistent Hashing scheme (Karger, 1997) to distribute keys among nodes, described in the following) over a subset of the total nodes, thus leading to the possibility to meet availability and fault tolerance requirements without the drawbacks that are endemic to the full replicated mode. Upon the commit of a transaction, only the nodes which store keys affected by it are involved in the two-phase commit.

High level architecture

In this section we will provide some details about the inner structure of Infinispan, describing the main building blocks that are at its core.

At the highest level of abstraction, the main components are the following: the Commands, which are objects that encapsulate the logic of the methods invoked by the user on the Cache; the Visitors, which are intended to "visit" the commands in order to execute them; special Visitors are the Interceptors, which act as a chain and interact with the subsystems which are affected by a command; the Managers, which are responsible for managing the subsystems.

Figure 2. Architectural Overview of Infinispan.
(InfinispanArchitectureOverview.tif)

The main building blocks of the Infinispan architecture are shown in Figure 2, and a brief description for each of them is provided in the following. We start by describing the modules that are common to any operational mode of Infinispan (which we call Core modules). Next we describe, in an incremental fashion, the additional modules that come into play in the full and partial replication modes.

Core Modules

1. DataContainer: it embeds a ConcurrentHashMap which ultimately stores the <key,value> pairs handled by users. Infinispan gives to the user the possibility to define a lifespan for the entries, after which they are removed from the data container. Moreover it supports several eviction algorithms (FIFO, LRU, LIRS (JBoss/Red Hat, 2010)) in order to avoid extreme usage of the memory heap. Evicted entries are completely removed, coherently with their lifespan, or stored through the registered CacheLoader, if any. Passivation of entries is also supported.
2. CommandFactory: it is responsible for mapping methods invoked by the application into commands which are visited by the Interceptors.
3. EntryFactory: it is responsible for wrapping DataContainer's entries when accessed. This component interacts with TxInterceptor, which is responsible for implementing the concurrency control algorithms that enforce the isolation level specified by the user.
4. LockManager: it deals with all aspects of acquiring and releasing locks for cache entries. It can support both lazy locking mode, meaning that upon a write the lock is taken only locally, postponing the system-wide lock acquisition until the prepare phase, and eager locking, in which a lock is taken immediately on all Infinispan's replica when a request is issued. As already mentioned, lock contentions are solved through timeout and in the case of multiple transactions waiting for the same lock fairness is not guaranteed. The Interceptor responsible for interacting with this component is the LockingInterceptor.
5. Transaction Manager: it allows to define transactions' boundaries. Infinispan is shipped with a simple DummyTransactionManager, but any JTA compliant transaction manager can be used. Infinispan provides also a XAAdapter class which implements the XAResource interface in order to support distributed transactions among heterogeneous resources.
6. InterceptorChain: it is the entry point for any command to be performed. It triggers the visit of the command from all the registered interceptors.
7. ComponentRegistry: it acts as a lean dependency injection framework, allowing components and managers to reference and initialize one another.

Additional modules involved in full replication mode:

1. Transport: it provides a communication link with remote caches and allows remote caches to invoke commands on a local cache instance. This layer ultimately relies on JGroups, to implement lower level's Group Communication Service's abstractions (such as view synchrony, failure detection, reliable broadcast, etc (Chockler, Keidar, & Vitenberg, 2001)).
2. CommandAwareDispatcher: it's a JGroups (Bela Ban/Red Hat, 2002) RPC dispatcher which knows how to deal with replicated commands.
3. InboundInvocationManager: it is a globally-scoped component which is able to reach named caches and trigger the execution of commands from remote ones by setting up the interceptor chain for them.
4. RpcManager: provides a mechanism for communicating with other caches in the cluster, by formatting and passing requests down to the registered Transport. This component dialogues directly with the ReplicationInterceptor, which triggers the re-execution of transactions via remote procedures calls.
5. StateTransferManager: it handles generation and application of state on the cache.

Additional modules involved in partial replication mode:

DistributionManager: it is responsible for both handling the correspondence between any data entry and its owner Infinispan node, as well as the caching policy of entries that are retrieved from remote instances when locally accessed. It communicates with the DistributionInterceptor, which is mutually exclusive with respect to the Replication one.

To determine which instances store which entries, the DistributionManager exploits a Consistent Hashing algorithm (Karger, 1997): both keys and caches are hashed onto a logical circle and each cache contains all keys between itself and the next clockwise one on the circle. This mechanism allows to detect the owner of an entry without additional communication steps and is extremely efficient since upon the leave/join of a node, only a reduced set of entries has to be redistributed, involving in this operation only the nodes which are logically adjacent to the one that has just left/joined the system.

This component is also responsible for managing a level one cache aimed at temporary storing entries which are not a burden, coherently with the consistent hashing, of a specific instance. If a transaction requests an entry that is not local to the node, a remote procedure call is issued, aimed at retrieving and storing it in the level one cache. DistributionManager is responsible for defining eviction policies for those entries.

Moreover, the DistributionManager module is in charge of removing from the DataContainer entries which a node is no longer responsible for (e.g. upon the join of a new instance) or of moving them to the level one cache.

Extensions to support complex dynamic reconfigurations

In order to deal with runtime reconfigurations, Cloud-TM enriches Infinispan's architecture with a Reconfiguration Manager. The Reconfiguration Manager is designed to communicate with the Autonomic Manager via JMX and, when a reconfiguration request is triggered, it orchestrates the switch operations of the various components via the native Infinispan's visitor-based design pattern.

A subsystem that wants its internals to be dynamically tuned must implement a Reconfigurable interface, exposing methods supporting the specification of generic reconfiguration commands, and returning a future (Goetz et al., 2006) associated with the outcome of the requested reconfiguration operation.

By exploiting the future semantics, we allow the invocation to the methods that encapsulate the reconfiguration logics to return asynchronously. This allows reconfigurations of modules that are not subject to specific causality constraints to be executed in parallel, reducing the total reconfiguration latency.

The final commit of the switch from a configuration to another one is issued by the Reconfiguration Manager, which can also act as a system-wide synchronization element to guarantee that, if the configuration switch has a global scope, then all Infinispan replicas have reached agreement on it before declaring the reconfiguration phase concluded. More in detail, upon the request for a reconfiguration, every subsystem involved in the reconfiguration has to perform modifications on its internal state and on its relevant fields on the Infinispan's Configuration object, which stores all state information concerning the configuration of a given instance of Infinispan.

At its core, a ConfigInfo object is a set of String and Objects: the Strings are the names of the methods exposed by the Configuration and are invoked via Reflection; the Objects can represent the parameters for such methods or any other element needed by a subsystem to perform the reconfiguration. The semantics of an Object as well as its specific class are determined at runtime, based both on the method currently invoked via Reflection and the specific logic implemented by the Reconfigure method.

Thanks to the Reconfiguration Manager, the above described architecture also encompasses the possibility to orchestrate complex reconfiguration operations that would be difficult to split into several, component-specific minor reconfigurations. For instance, this design pattern allows to modify the interceptor chain at run-time. The latter can therefore be seen as both the means through which reconfiguration commands are spread in the system, and as a reconfigurable object itself. A practical example for this scenario is the switch from Replicated to Distributed mode, which entails the substitution of the ReplicationInterceptor with the DistributionInterceptor.

Figure 3 - WPM Architectural Organization (WPMArchitecturalOrganization.tif)

Autonomic Manager

The Autonomic Manager (AM) is the component of the Cloud-TM platform that is in charge of automating the elastic scaling of the Data Platform, as well as of orchestrating the self-optimizing strategies to dynamically reconfigure the data distribution and replication mechanisms. The AM aims at minimizing the infrastructure cost by minimizing the amount of resources required by the applications and optimizing their usage, avoiding at the same time Quality of Service (QoS) violations.
Figure 1 from Section “Overview of the Cloud-TM Platform” depicts the AM architecture. Its topmost layer exposes an API allowing the specification and negotiation of QoS requirements and budget constraints. In order to implement self-optimization strategies, the AM relies on the Workload & Performance Monitor (WPM) component for sensing the execution environment. Figure 3 depicts the WPM. WPM collects information from several heterogeneous components of the platform. Gathered statistics refer to both the utilization of hardware resources and performance metrics at data platform level. The streams of statistical data are used by another component of the AM, the Workload Analyzer (WA). The WA implements workload and resource demand characterization and prediction functionalities by relying on aggregated and filtered data views. The WA also offers the possibility to set alerts to be triggered when specific pre-defined conditions are met (e.g. QoS violations).
Finally, another component, namely the Adaptation Manager (AdM), is in charge of actuating the platform reconfiguration throughout the functionalities provided by the WA and by using both off-line adaptation policies (e.g. based on alerts) and on-line performance forecasts.

Workload & Performance monitor

The Workload and Performance Monitor (WPM) provides audit data for both infrastructure resources and platform (or application) level components in an integrated manner. An important feature of WPM is that it does not target any specific platform or application. Instead, it is flexible and adaptable, so to allow integration with differentiated platform/application types. On the technological side, the development of WPM leverages on (a) Lattice, a monitoring framework tailored for cloud infrastructures, which has been developed in the context of the RESERVOIR EU project, and on (b) JMX, a monitoring framework suited for the audit of Java-based components.
The WPM, whose architecture is shown in Figure 3, has been defined according to the need for supporting statistical data gathering (SDG) and logging (SDL) functionalities.
The SDG functionality maps onto an instantiation of a flexible, fault-tolerant and easy to reconfigure system that is able to interconnect all the components of the monitoring infrastructure. In order to be more general and maximize the coverage of the possible configurations of a distributed system, the elements belonging to the monitored infrastructure, such as Virtual Machines (VMs), can be logically grouped, and each group will entail per-machine probes targeting two types of resources: (a) hardware/virtualized and (b) logical ones. Statistics related to the first kind of resources could be directly collected over the operating system (OS), or via an OS-decoupled library, while statistics related to logical resources (e.g. the data-platform) are collected at the application level. The latter approach does not necessarily require an instrumentation of the monitored application but, relying on the JMX Java framework, WPM is able to directly query any Java applications. Also, it can be exploited to build a Java wrapper for non-Java applications/components. The data collected by the probes is sent to the producer component via the facilities natively offered by the interconnection systems. Each producer is coupled with one or many probes and is responsible of managing them. The consumer is a component that receives the data from the producers, via differentiated messaging implementations, which could be selected on the basis of the specific system deployment. A classical network configuration is composed by a LAN-based clustering scheme such that the consumer is in charge of handling one or multiple groups of machines belonging to

the same LAN. Anyway, the number of consumers is not meant to be fixed. Conversely, it can be scaled up/down depending on the amount of instantiated probes/producers. Overall, the consumers can be instantiated as centralized or distributed processes. Beyond collecting data from the producers, the consumer is also in charge of performing a local elaboration aimed at producing a suited stream representation to be provided as the input to the Log Service component, which is in charge of supporting the SDL functionality.

An easy way to implement the consumers is to decide to exploit their locally available file systems, to temporarily keep the stream instances to be sent towards the Log Service component. The functional block which is responsible of the interaction between SDG and SDL is the so-called optimized-transmission service. This can rely on top of differentiated solutions depending on whether the instance of SDL is co-located with the consumer or resides on a remote network. Generally speaking, it can use a technology such as SFTP or a locally-shared File System to exchange data. Also, stream compression schemes can be actuated to optimize both latency and storage occupancy. The Log Service is the logical component responsible for storing and managing all the gathered data. It must support queries from any external application so to expose the statistical data to subsequent processing/analysis tasks. The Log Service could be implemented in several manners, in terms of both the underlying data storage technology and the selected deployment (centralized vs. distributed). As for the first aspect, different solutions could be envisaged in order to optimize access operations depending on, e.g. suited tradeoffs between performance and flexibility. This is also related with the data model ultimately supported by the Log Service, which might be a traditional relational model or, alternatively, a <key,value> model. Further, the Log Service could maintain the data onto a stable storage support or within volatile memory, for performance and reliability tradeoffs. The above aspects could depend on the functionalities/architecture of the application that is responsible for analyzing statistical data, which could be designed to be implemented as a geographically distributed process in order to better fit the WPM deployment (hence taking advantage from data partitioning and distributed processing).

Workload Analyzer

The WA is placed between the WPM and the AdM. The WA bears the following responsibilities:

- Data aggregation and filtering;
- Workload and resource demand characterization;
- Workload and resource demand prediction;
- QoS monitoring and alert notification.

As for technologies at the base of the construction of the WA, RHQ plays a central role. RHQ is a JBoss open source product for systems management. It can interface with several products and platforms and provides various management functionalities, such as monitoring, alerting, remote configuration and operations execution. It leverages on a flexible server/agent architecture. An agent allows the server to connect to a managed resource. The server gets monitoring data by agents, stores it in a database and provides a friendly use interface for data analysis and visualizations. Furthermore, server allows users to trigger operations for the remote management. Agents can be extended by means of plug-ins to add new functionalities. Let us now go through details in relation to the functionalities offered by the WA.

Data aggregation and filtering

The logical dislocation of computational nodes within the distributed system assumed by the WA is inherited by the one of the Autonomic Manager. In fact, the WA exposes facilities to support aggregation and filtering of incoming monitoring data streams, in order to represent an organization of the components that can be monitored, based on the presence of logical groups that aggregate homogeneous resources. To do that, the WA exploits the advanced grouping functionalities provided by RHQ. Summarizing, the presence of logical groups serves a twofold purpose:

- defining which access permission are applied to the monitored resources;
- providing a way to view aggregate data and perform actions across all group members.

RHQ enables flexible group membership policies, which support not only the manual addition of resources to groups, but also the definition of regular expressions, called DynaGroups, that maintain group membership in a dynamic fashion. Once groups are defined, it is possible to specify access control polices directly to the groups of resources, instead of individual resources. By using DynaGroups, the system administrator can effectively create dynamic ACLs (Access Control Lists) to lessen the burden of security maintenance, especially against large number of resources. Compatible groups (those composed entirely of the same type of resource, e.g. all Linux platforms) have additional features available to them, such as: group-wise availability; min, max, and average metrics across the group; aggregate events viewer; operations against all group members, either serialized or concurrent execution policies; fine-grained changes to connection properties and resource configuration across one or more members of the group.

Workload and resource demand characterization

The WA has also the capability to characterize the workload and resource demand of transactional applications deployed on the Cloud-TM platform. To do that, the WA acquires a large amount of statistical information from the various layers of the Cloud-TM platform. In order to profile the behavior of transactional applications, the WA is able to compute complex statistics on the basis of data collected using a number of probes scattered across several sub-systems of the Cloud-TM's data grid. These data are externalized using JMX interfaces in order to permit their monitoring via the WPM system.

These statistics can be classified into high-level and low-level statistics. To the high-level statistics belong all the statistical data related to the identification of hot spot data items, and all the statistics aimed at identifying the maximum degree of data parallelism for an application. For what concerns hot-spot identification, the main statistic provided is called “top-k” keys (where k is a parameter that is dynamically configurable). It derives form tracing the data accessed by the applications when:

- an update operation is issued;
- a read operation is issued, either remotely or locally - thus requiring or not a remote interaction with another node during transaction execution;
- a lock, causing no contention, contention, or abort of a transaction is issued.

This information is critical in a distributed environment for managing and optimizing the data placement and the concurrency management scheme.

The outcome of “top-k” statistics can be used for managing and optimizing the data placement and the concurrency management schemes within the data grid, and it is extremely valuable for the automatic and/or human-driven tuning of these performance-critical modules of the system. In order to minimize overheads, the process of top-k identification relies on a recently proposed algorithm coming from the world of data stream analysis. The algorithm has been presented in (Metwally, Agrawal, & Abbadi, 2006) (and implemented by the stream-lib open-source project (Clearspring Technologies, 2012). It offers several advantages in terms of limited (constant) memory space and great performance.

In addition, for what concerns the identification of maximum degree of data parallelism exposed by the applications, the WA computes an innovative metric, which is called Application Contention Factor (Didona, Romano, Peluso, & Quaglia, 2011).

The low-level statistics provide a detailed characterization of the performance and costs of the main subsystems involved in the processing of transactions along its life-cycle. These include both statistics (mean, and percentiles) on metrics typically used in SLAs (for instance, transaction execution time) and statistics useful for modeling purposes, such as the latency experienced by transactions along their various execution stages, the frequency of different types (write vs. read) of transactions and of various contention-related events (e.g. successful vs. failed lock acquisition). Among these, two types of statistics are particularly noteworthy: the probability distribution of lock inter-arrival time and the percentiles of transaction execution times.

Workload and resource demand prediction

The WA offers functionalities for post-processing collected statistics in order to have an estimation of the future workload in terms of user behavior trends and resource demand. In order to be as general and powerful as possible, rather than embedding WA specific prediction models, it relies on the well-known R statistical engine (R-project, 2012). Using R, a client of the WA is able to process in whatever manner the statistics exposed by the WA and re-inject the results again to the WA. This is made possible by exploiting the recently introduced REST APIs of RHQ, which allow exporting the statistical data gathered from the monitored platform as time-series encoded in a standard format, which of course could be parsed by R. Examples of these post-processing range from simple moving average to more complex 5% and 95% quartiles. This approach that permits to retrieve, process and then store again the results within the WA. As a final remark, the flexibility of this approach allows to use engines to post-process data that are not necessarily the mentioned R, but could be analytical models (Sanzo, Ciciani, Quaglia, Palmieri, & Romano, 2012) or machine learning tools (Couceiro, Romano, & Rodrigues, 2011; Couceiro, Romano, & Rodrigues, 2010).

QoS monitoring and alert notification

The WA provides an advanced QoS monitoring and alert notification engine, leveraging on the functionalities offered by RHQ (JBoss/Red Hat, 2012a). The latter engine is designed to provide proactive notifications about events happening throughout the monitored platform. These events can be resources becoming unavailable, specific values for metrics being collected, resource configuration being changed, operations being executed, or even specific conditions found by parsing log events. Each stream of data within the WA passes through the alerts processing engine. Here, the data can be transformed, filtered, or even correlated with data from other parts of the system. Users have full control over what they want to be notified about, and the engine keeps a full audit trail of the conditions that have triggered alerts to fire. The alerts subsystem provides a wealth of different options for being notified proactively about potential issues in the system. As a result, it supports a breadth of different configuration options that allow for deriving very specific and customized semantics.

Figure 4 – Adaptation Manager Organization.
(AdaptationManagerOrganization.tif)

Adaptation Manager

The diagram in Figure 4 provides an architectural overview of the Adaptation Manager. This module is in charge of defining the global self-optimization strategy of the Cloud-TM platform and of orchestrating the platform's reconfiguration process by coordinating the tuning of the ecosystem of components forming the Data Platform, and the provisioning resources from IaaS Cloud providers. The optimization of the Cloud-TM platform is managed by the Optimization Manager component, which takes as input the following data flows:

- the QoS/Cost constraints specified by the users via the QoS API;
- alert signals, performance/cost statistics and workload characterization information generated by the workload analyzer;
- the meta-data of the various tunable components of the Cloud-TM platform.

The key role of the Optimization Manager is to manage the life cycle (instantiate, interface and trigger the activation) and mediate the outputs of the set of component optimizers that will be developed throughout the project. The Cloud-TM Data Platform is in fact constituted by a complex ecosystem of components, each one associated with specific key performance indicators, utility functions and tunable parameters. Further, due to the layered architecture of the Data Platform, and to non-trivial interdependencies that exist among its constituting components, it is natural to expect that the configuration strategy of a given component, say "Comp", may strongly affect not only the efficiency of that same component, but also of the efficiency components that interact (directly or indirectly) with Comp.

Classic monolithic optimization approaches, which try to optimize the system as a whole by modeling the dynamics of all of its components using a single, unified approach, are extremely complex to use in a large and heterogeneous system such as the Cloud-TM platform. In order to master complexity, the Cloud-TM platform uses a "divide-et-impera" approach, which subdivides the global optimization task into a set of simpler local optimization sub-problems, focused on determining the optimal configuration policy for small subsystems of the Data Platform. The optimization process proceeds then in an iterative fashion, propagating the effects of the tuning of each component (such as shifts of the workload characteristics for other components, or alteration of the demands of shared system resources) along a chain that captures the existing mutual interdependencies among the Data Platform's components.

The Optimization Manager is the coordinator of this concerted optimization process, triggering its activation periodically, or upon reception of alert signals generated by the Workload Analyzer's module, and mediating the outputs (i.e. reconfiguration policies) of the various local optimizers in order to identify a globally optimal reconfiguration strategy (or at least as close as possible to the global optimum).

An important research line that we are pursuing is to investigate how to combine optimization techniques of different nature, including analytical (Sanzo, Ciciani, Quaglia, Palmieri, & Romano, 2012) and simulative (Miller & Griffeth, 1991) models, as well as approaches based on machine learning (Couceiro et al., 2010) and control theory (Wang, Zhu, & Singhal, 2005). Each of these methodologies comes in fact with its pros and cons, and results particularly adequate to control different classes of systems.

Model-based performance forecast approaches (e.g. relying on analytical or simulative techniques) typically demand very short training time, but can only be employed if the model developer has a detailed knowledge of the system's internals and dynamics. On the other hand, model-free control methods, which treat the system to control as black-boxes and learn their dynamics using statistical techniques, are more generic and robust than model-based techniques (whose accuracy is affected by the accuracy of the employed system's model), but they are better suited to deal with low-dimensional control problems, as their learning time typically grows exponentially with the number of variables to be monitored/controlled.

Some of the results achieved along this research direction include the work in (Romano & Leonetti, 2012), where analytical modeling is used to bootstrap the knowledge base of an on-line machine learner aimed to self-tune the message packing level (Friedman & Hadad, 2006) of a Group Communication System (Guerraoui & Rodrigues, 2006). This technique combines the best of the worlds: boosting, on one hand, the learning phase of the machine learner, while enhancing, on the other hand, the accuracy of the predictions initially generated by the analytical model.

Another relevant approach is represented by TAS (Didona et al., 2011), a system for automating the elastic scaling (i.e. the resource provisioning) of the Cloud-TM's in-memory transactional data grid. In this case, a divide-et-impera approach is taken, in which machine learning is used to forecast the response of the system in presence of different contention levels for physical resources (e.g. CPU, memory, network), whereas white-box analytical modeling is employed to forecast the effects of data contention among concurrent transactions. The black-box nature of machine learning techniques spares from the burden of explicitly modeling the interactions with system resources that would be otherwise needed using white-box, analytical models. This would be not only a time-consuming and error-prone task given the complexity of current hardware architectures. It would also constrain the portability of our system (to a specific infrastructural instance), as well as its practical viability in virtualized Cloud environments where users have little or no knowledge of the underlying infrastructure. On the other hand, analytical modeling allows us to address two well-known drawbacks of machine learning, namely its limited extrapolation power and lengthy training phase. By exploiting a-priori knowledge on the dynamics of data consistency mechanisms, analytical models achieves good forecasting accuracy even when operating in previously unexplored regions of the parameters' space. Further, by narrowing the scope of the problem tackled via machine learning techniques, analytical modeling allows to reduce the dimensionality of the machine learning input features' space, leading to a consequent reduction of training phase duration (Sutton & Barto, 1998).

Returning to the diagram in Figure 4, the Adaptation Manager will store the Data Platform component's optimizers within an apposite repository that will allow both the retrieval and update of optimizers logic

and data (e.g. training data-sets or other statistical information gathered by observing the results of previous self-tuning cycles).
The Adaptation Manager will include two additional building blocks, the Global Reconfiguration Manager and the TunableResource Abstraction Layer. The former will be in charge of coordinating global reconfiguration actions that entail orchestrating different layers of the Data Platform, such as provisioning new nodes from the IaaS provider to the Data Platform, and having them join the Transactional In-memory Data Grid. Note that our architecture for reconfiguration is inherently hierarchical. Acting from a privileged, centralized perspective, the Global Reconfiguration Manager will, in fact, be able to trigger reconfiguration of complex systems (such as the In-memory transactional data grid) which may, in their turn, implement non-trivial distributed protocols to complete their own reconfiguration.
The TunableResource Abstraction Layer will serve the purpose of hiding from the Global Reconfiguration Manager the heterogeneity of interacting with the various platform's components, which will encompass modules implemented using different technologies and remotely. Under the scenes, this module will encapsulate the logic to interact with the IaaS provider, in order to automate the resource provisioning process, and with the Data Platform's components, via the Control Plane of the Lattice framework. Note that, as shown in Figure 4, in order to achieve transparent interoperability with multiple IaaS Cloud providers, the AdM exploits the delta-cloud APIs (Apache Software Foundation, 2011), a recent open-source project that has recently been accepted into the Apache Software Foundation Incubator, which provides, in its turn, an abstraction layer towards multiple IaaS Cloud providers (including Amazon EC2/S3, GoGrid, OpenNebula, Rackspace Cloud Servers/Cloud Files).

FUTURE RESEARCH DIRECTIONS

The Cloud-TM project is, at the moment of writing, entering its third and last year, and the consortium is working intensively to materialize the vision outlined in this book chapter. In this section we overview some of the most relevant research lines that are being pursued in this stage. Note that, while we expect that these research lines will yield to tangible results that will be incorporated in the final prototype of the Cloud-TM platform. On the other hand, some of these research problems are so broad that it is unlikely that they will be fully explored by the end of the project. Conversely, we believe that the Cloud-TM project will blaze new trails that, we hope, will be pursued in future also by other researchers active in the area of Cloud computing.

Performance Forecasting Models for Very Large Scale DSTM systems. As mentioned previously, the TAS system (Didona et al., 2011) represents a significant leap towards the fulfillment of the objective of automating the resource provisioning process in a DSTM system. TAS, however, addresses the scenario of fully replicated DSTM, in which each node maintains a full replica of the DSTM's state. This class of data consistency mechanisms is optimized for small-medium scale platforms, e.g. composed of 10-20 nodes.
However, the cost of propagating the state updates generated by a transaction across all the nodes in the system becomes quickly prohibitive as the system scale grows. Large scale DSTM systems, spanning hundreds or thousands of nodes, rely on inherently more scalable, so called *genuine* (Peluso, Romano, & Quaglia, 2012; Peluso, Ruivo, Romano, Quaglia, & Rodrigues, 2010), replication schemes, which ensure a limited degree of replication (hence, bounding the cost of replication even in large scale systems) and avoid the reliance on any centralized component (hence, avoiding bottlenecks in the system design).
Designing solutions capable of forecasting the effects of contention on data and physical resources on the performance of very large scale DSTM platforms represents an important line of research, which is, to the best of our knowledge, still unexplored in literature.

Self-tuning of data placement. As discussed above, a key choice when deploying a DSTM is whether to adopt a full or a partial replication model. The latter is more scalable, but, on the down side, it incurs in the costs of remote reads (as not all data are available at all nodes) during transactions' execution. Thus, a key factor affecting performance of partially replicated systems is associated with how data is distributed across the nodes of the system. This is an issue that is being intensively studied in the context of the Cloud-TM project, with the aim of designing self-tuning data placement algorithms capable of exploiting application's locality to minimize the costs of remote data fetching.

In particular, we are exploring a crucial trade-off in the design space of data placement algorithms that is associated with the granularity on whose basis data placement algorithms work. On one hand, algorithms that operate with the granularity of a single data instance have maximum flexibility, and have, consequently, the maximum potentiality of maximizing data access locality. Unfortunately, they also incur in the highest book-keeping costs, as they need to explicitly keep track of the placement of potentially a very high number of data. On the other hand, algorithms operating at the granularity of the data item type (i.e. object-type in an object-oriented data platform like Cloud-TM) incur in significantly lower book-keeping overheads, hence resulting feasible also in big data scenarios. Unfortunately, however, they also have a much more limited flexibility, which may lead to suboptimal data placement solutions.

Driven by the requirements of high scalability, the solutions developed so far in the project (Garbatov & Cachopo, 2012; Garbatov & Cachopo, 2011a; Garbatov & Cachopo, 2011b; Garbatov & Cachopo, 2011c) have been based on the object-type approach. An interesting area that is being currently explored in the Cloud-TM project is the design of instance-based techniques that exploit probabilistic techniques in order to avoid maintaining exact data placement information. The idea being pursued is to exploit space-efficient encoding techniques (such as Bloom filters (Broder & Mitzenmacher, 2005) or associative neural memories (Hassoun, 1993) to achieve the flexibility of instance-based approaches, while drastically abating their book-keepings costs, in order to achieve scalability levels comparable to those of type-based systems.

Self-tuning of Replication Protocols. One of the key innovative ideas explored in the Cloud-TM project is to pursue optimal efficiency by adapting dynamically the algorithms to enforce data consistency depending on fluctuations of the workload and of the scale of the DSTM platform. Polycert (Couceiro et al., 2011) represented a first important step towards the achievement of this goal. However, there are still important aspects that need to be further investigated. Polycert supports the coexistence among multiple Atomic Broadcast based (Guerraoui & Rodrigues, 2006) certification protocols, relying on machine learning techniques to determine the optimal protocol to use on a per-transaction basis. On the other hand, the design space of replication protocols for transactional systems is very wide (Couceiro, Romano, & Rodrigues, 2011), and the problem of supporting efficient re-configurations among arbitrary replication protocols remains an important open problem that we are currently addressing in our most recent efforts.

Another interesting research direction in this area is related to how to automatically identify which data consistency protocols to use given the current load/scale scenario. Machine learning techniques, as used in Polycert, are attractive due to their black box nature, which makes them capable of handling a wide range of alternative options in a relatively simple way. On the other hand, precisely due to their black box nature, they have normally relatively little extrapolation power, i.e. they have a very limited ability to predict system's performance in presence of workloads not observed during the training phase. Due to their white box nature, analytical models do not suffer of the above limitation, but they are normally much more expensive to design and validate. Exploring the, possibly combined, usage of techniques relying on statistical learning and analytical models is another intriguing research area that we plan to explore in our future work.

CONCLUSION

Cloud-TM is an ongoing European project aimed at designing and implementing an elastic, self-tuning transactional store for the cloud. Specifically, the Cloud-TM platform aims at dynamically adjusting the

amount of resources acquired from the underlying IaaS provider depending on possible fluctuations of incoming workload and user SLAs. In parallel, the Cloud-TM platform will self-tune its configuration in order to ensure maximal efficiency across scenarios characterized by different platform scales and heterogeneous workloads. From a programming perspective, the API provided by the Cloud-TM platform will be based on the abstraction of transactional memory, and will spare the programmers from dealing with the inherent complexities associated with the development of large scale distributed applications.
This chapter has overviewed the general architecture of the Cloud-TM platform. In this process we have illustrated the key motivations and challenges that were accounted for during the design of the platform and presented some of the results achieved so far within the project.

REFERENCES

Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. IEEE Data Eng. Bulletin, 32(1), 3-12.

Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., & Karamanolis, C. (2007). Sinfonia: a new paradigm for building scalable distributed systems. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (pp. 159–174). New York, NY: ACM.

Alonso, G., & Casati, F., & Kuno, H., & Machiraju, V. (2004). Web Services: Concepts, Architectures and Applications. Berlin, Germany: Springer.

American National Standards Institute (1992). ANSI SQL-92 Specification, Document Number: ANSI X3.135-1992. Retrieved November 30, 2012, from http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

Amir, Y., Danilov, C., & Stanton, J. (2000). A low latency, loss tolerant architecture and protocol for wide area group communication. In Proceedings of the International Conference on Dependable Systems and Networks (pp. 327-336). New York, NY: ACM.

Apache Software Foundation (2011). δ-cloud. Retrieved November 30, 2012, from http://deltacloud.apache.org.

Bela Ban/Red Hat (2002). JGroups - The JGroups Project. Retrieved November 30, 2012, from www.jgroups.org.

Birman, K. P. (1993). The process group approach to reliable distributed computing. Communications of the ACM, 36(12), 37–53.

Bocchino, R.L., & Adve, V.S., & Chamberlain, B.L. (2008). Software transactional memory for large scale clusters. Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP) (pp. 247–258). New York, NY: ACM.

Broder, A., &; Mitzenmacher, M. (2005). Network Applications of Bloom Filters: A Survey. Internet Mathematics, 1 (4), 485–509.

Cachopo, J. & Rito-Silva, A. (2006). Combining software transactional memory with a domain modeling language to simplify web application development. In Proceedings of the 6th international conference on Web engineering (pp. 297–304). New York, NY: ACM.

Cachopo, J. (2007). Development of Rich Domain Models with Atomic Actions. Unpublished doctoral dissertation, Technical University of Lisbon, Portugal.

Carvalho, N., & Romano, P., & Rodrigues, L., (2010). Asynchronous Lease-based Replication of Software Transactional Memory. In Proceedings of the ACM/IFIP/USENIX 11th Middleware Conference (Middleware) (pp 376-396). New York, NY: ACM.

Carvalho, N., Cachopo, J., Rodrigues, L., & Rito-Silva, A. (2008). Versioned transactional shared memory for the fenixedu web application. In Proceedings of the Second Workshop on DependableDistributed Data Management (pp 15-18). New York, NY: ACM.

Chockler, G., Keidar, I., & Vitenberg, R. (2001). Group communication specifications: a comprehensive study, ACM Comput. Surveys, December 2001. ACM Computing Surveys, 33(4), 427-469.

Clayman, S., & Galis, A., & Mamatas, L. (2010). Monitoring virtual networks with lattice, Paper presented at Network Operations and Management Symposium Workshops, Osaka, Japan.

Clearspring Technologies (2012). The stream-lib library. Retrieved November 30, 2012, from https://github.com/clearspring/stream-lib.

Codd, E. F. , & Codd, S. B., & Salley, C. T (1993)., Providing OLAP (On-Line Analytical Processing) to User-Analysis: An IT Mandate, E. F. Codd & Associates.

Couceiro, M., & Romano, P., & Rodrigues, L. (2010). A Machine Learning Approach to Performance Prediction of Total Order Broadcast Protocols. In Proceedings of the 2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (pp 184-193). Washington, DC: IEEE Computer Society.

Couceiro, M., & Romano, P., & Rodrigues, L., (2011). PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids. In Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware (pp 309-328). Berlin, Germany: Springer-Verlag.

Couceiro, M., Romano, P., & Rodrigues, L. (2011). Towards Autonomic Transactional Replication for Cloud Environments. In D. Petcu and J. Poletti (Ed.), European Research Activities in Cloud Computing. Cambridge, UK: Cambridge Scholars Publishing.

Couceiro, M., Romano, P., Carvalho, N., & Rodrigues, L. (2009). D2stm: Dependable distributed software transactional memory. In Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing (pp. 307-313). Washington, DC: IEEE Computer Society.

DeWitt, D., & Stonebraker, M. (2009). Mapreduce: A major step backwards. Retrieved November 30, 2012, from http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.

Di Sanzo, P., Ciciani, B., Quaglia, F., Palmieri, R., & Romano, P., (2012) On the Analytical Modeling of Concurrency Control Algorithms for Software Transactional Memories: the Case of Commit-Time-Locking, Performance Evaluation 69(5), 187-205.

Didona, D., & Romano, P., & Peluso, S., & Quaglia, F. (2011). Transactional Auto Scaler: Elastic Scaling of In-Memory Transactional Data Grids. In The 9th International Conference on Autonomic Computing (ICAC 2012) (pp. 17-21). New York, NY: ACM.

Felber, P., Fetzer, C., & Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (pp. 237–246). New York, NY: ACM.

Friedman, R., & Hadad, R., (2006). Adaptive batching for replicated servers. In Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (pp. 311–320). Washington, DC: IEEE Computer Society.

Garbatov, S., & Cachopo. J., (2011b). Software Cache Eviction Policy based on Stochastic Approach. In Proceedings of the Sixth International Conference on Software Engineering Advances.

Garbatov, S., & Cachopo. J., (2011c). Optimal Functionality and Domain Data Clustering based on Latent Dirichlet Allocation. In Proceedings of the Sixth International Conference on Software Engineering Advances. XPS.

Garbatov, S., & Cachopo. J., (2011a). Data Access Pattern Analysis and Prediction for Object-Oriented Applications. S. Garbatov and J. Cachopo. INFOCOMP Journal of Computer Science, 4(10) 1-14.

Garbatov, S., & Cachopo. J., (2012). Decreasing Memory Footprints for Better Enterprise Java Application Performance. In Proceedings of the 23rd International Conference on Database and Expert Systems Applications (pp. 430-437). Berlin, Germany: Springer.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java Concurrency in Practice. Boston, MA: Addison Wesley.

Gramoli, V., Harmanci, D., & Felber P. (2008). Toward a Theory of Input Acceptance for Transactional Memories. In Proceedings of the 12th International Conference On Principles Of Distributed Systems (pp. 527-533). Berlin, Germany: Springer.

Gray, J., & Reuter, A. (1993), Transaction Processing: Concepts and Techniques. Oxford: Elsevier Lt.

Guerraoui, R. & Kapalka, M. (2008). On the Correctness of Transactional Memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp.175-184). New York, NY: ACM.

Guerraoui, R., & Rodrigues, L., (2006). Introduction to Reliable Distributed Programming, Berlin, Germany: Springer.

Guerraoui, R., & Kapałka, M., (2008). On the Correctness of Transactional Memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), (pp 175-184). New York, NY: ACM.

Guerraoui, R., Herlihy, M, & Pochon B. (2005). Polymorphic Contention Management. Proceedings of the nineteenth International Symposium on Distributed Computing (pp. 303-323). Berlin, Germany: Springer.

Guyon, I., & Elisseeff, A. (2003) An introduction to variable and feature selection. The Journal of Machine Learning Research, vol. 3, 157–1182.

Hadoop Wiki (2012). Hadoop Map/Reduce Data Processing Benchmarks. Retrieved November 30, 2012, from http://wiki.apache.org/hadoop/DataProcessingBenchmarks.

Harris, T. & Fraser, K. (2003). Language support for lightweight transactions. ACM SIGPLAN Notices, 38(11), 388–402.

Hassoun, M. H. (1993). Associative Neural Memories: Theory and Implementation. Oxford University Press, USA.

Herlihy, M., Eliot, J., & Moss, B. (1993). Transactional memory: Architectural support for lock-free data structures. In Proceedings of the International Symposium on Computer Architecture (pp. 289-300). New York, NY: ACM.

Herlihy, M., Luchangco, V., Moir, M., & Scherer, III, W. N. (2003). Software transactional memory for dynamic-sized data structures. Proceedings of the twenty-second annual symposium on Principles of distributed computing (pp. 92–101). New York, NY: ACM.

Hibernate (2009). Hibernate. Retrieved November 30, 2012, from http://www.hibernate.org.

iBATIS (2009). iBATIS. Retrieved November 30, 2012, from http://ibatis.apache.org.

Java Community process (2001). Java Caching API. Retrieved November 30, 2012, from http://jcp.org/en/jsr/summary?id=107.

JBoss/Red Hat (2001). INFINISPAN - Open Source Data Grids. Retrieved November 30, 2012, from http://www.jboss.org/infinispan.

JBoss/Red Hat (2010). Infinispan eviction, batching updates and LIRS. Retrieved November 30, 2012, from http://infinispan.blogspot.com/2010/03/infinispan-eviction-batching-updates.html.

JBoss/Red Hat (2012a). RHQ project. Retrieved November 30, 2012, from http://www.rhq-project.org.

JBoss/Red Hat (2012b). Teiid. Retrieved November 30, 2012, from http://www.jboss.org/teiid.

Jeffrey, D., & Sanjay, G., (2008). MapReduce: simplified data processing on large clusters, Communications of the ACM, 51(1), 107-113.

Kalyvianaki, E., Charalambous, T., & Hand, S. (2009). Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In Proceedings of the 6th International Conference on Autonomic Computing (pp. 117-126). New York, NY: ACM.

Karger, D., & Lehman, V, & Leighton, T., & Panigrahy, R., & Levine, V., & Lewin, D.. (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the Twenty-ninth annual ACM symposium on Theory of computing (pp. 654–663). New York, NY: ACM.

Keleher, P., Cox, A. L., & Zwaenepoel, W. (1992). Lazy release consistency for software distributed shared memory.In Proceedings of the 19th annual international symposium on Computer architecture (pp. 13–21). New York, NY: ACM.

Kotselidis, C., & Ansari, M., & Jarvis, K., & Lujan, M., Kirkham, C., Watson, I. (2008). DiSTM: A software transactional memory framework for clusters. Proceedings of the International Conference on Parallel Processing (ICPP) (pp. 51–58). Washington, DC: IEEE Computer Society.

Manassiev, K., & Mihailescu, M., & Amza, C., (2006). Exploiting distributed version concurrency in a transactional memory cluster. In Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP) (pp. 198–208). New York, NY: ACM.

Metwally, A., & Agrawal, D., & Abbadi, A. E. (2006). An integrated efficient solution for computing frequent and top-k elements in data streams. ACM Transactions on Database Systems, 31(3), 1095–1133.

Miller, J. & Griffeth, N., (1991). Performance modeling of database and simulation protocols: design choices for query driven simulation. In Proceedings of the 24th annual symposium on Simulation (pp. 205-216). Washington, DC: IEEE Computer Society.

Miranda, H., Pinto, A., & Rodrigues, L. (2001). Appia, a flexible protocol kernel supporting multiple coordinated channels.In Proceedings of the 21st InternationalDST Conference on Distributed Computing Systems (pp. 707–710). Washington, DC: IEEE Computer Society.

Moir, M. (1997). Transparent support for wait-free transactions.In Proceedings of the 11th International Workshop on Distributed Algorithms (pp. 305–319). Berlin, Germany: Springer.

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In Proceedings of the International Conference on the Management of Data (pp. 1099–1110). New York, NY: ACM.

Oracle (2012). Introduction to the Java Persistence API (Chapter 32 of The Java EE 6 Tutorial). Retrieved November 30, 2012, from http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

Pacitti, E., Coulon, C., Valduriez, P., & Özsu, M. T. (2005). Preventive replication in a database cluster. Distributed and Parallel Databases, 18(3),223–251.

Peluso, S., Romano, P., & Quaglia, F., (2012). SCORe: a Scalable One-Copy Serializable Partial Replication Protocol. In ACM/IFIP/USENIX 13th International Middleware Conference (to appear).

Peluso, S., Ruivo, P., Romano, P., Quaglia, F., & Rodrigues, L., (2010). When Scalability Meets Consistency: Genuine Multiversion Update Serializable Partial Data Replication. In Proceeding of the IEEE 32nd International Conference on Distributed Computing Systems (pp 455-465). Washington, DC: IEEE Computer Society.

Ramadan, H. E., Rossbach C. J.,. Porter D. E., Hofmann, O. S., Bhandari, A., & Witchel, E. (2007). MetaTM/TxLinux: transactional memory for an operating system. In Proceedings of the 34th annual international symposium on Computer architecture (pp. 92-103). New York, NY: ACM.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., & Kozyrakis, C. (2007). Evaluating mapreduce for multi-core and multiprocessor systems. Proceedings of the International Symposium on High-Performance Computer Architecture (pp. 13–24). Washington, DC: IEEE Computer Society.

Reservoir Consortium (2012). Reservoir, Resources and Services Virtualization without Barriers. Retrieved November 30, 2012, from http://www.reservoir-fp7.eu/.

Riegel, T., Fetzer, C., & Felber, P. (2008). Automatic data partitioning in software transactional memories. In Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (pp. 152-159). New York, NY: ACM

Romano, P. & Leonetti, M., (2012). Poster: selftuning batching in total order broadcast via analytical modelling and reinforcement learning. ACM SIGMETRICS Performance Evaluation Review - Special Issue on IFIP PERFORMANCE 2011- 29th International Symposium on Computer Performance, Modeling, Measurement and Evaluation, 39(2), 77-77.

R-project (2012). The R-project. Retrieved November 30, 2012, from http://www.r-project.org.

Shavit, N., & Touitou, D. (1995). Software transactional memory. In Proceedings of the Symposium on Principles of Distributed Computing (pp. 204-213). New York, NY: ACM.

Shpeisman, T., Adl-Tabatabai, A, Geva, R., Ni, Y., & Welc, A.(2009). Towards Transactional Memory Semantics for C++. In Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures (pp. 49-58). New York, NY: ACM.

Sonmez, N., Cristal, A., Harris, T., Unsal, O. S., & Valero, M. (2009). Taking the heat off transactions: dynamic selection of pessimistic concurrency control. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-10). Washington, DC: IEEE Computer Society.

SUN Microsystems (1999). Java Transaction API (JTA). Retrieved November 30, 2012, from http://www.oracle.com/technetwork/java/javaee/jta/index.html.

SUN Microsystems (2002). Java Management Extensions (JMX) v1.2 Specification. Retrieved November 30, 2012, from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html.

Sun MicroSystems (2003). Java 2 Platform Enterprise Edition Specification, v. 1.4, final release edition, 2003. Retrieved November 30, 2012, from http://www.jcp.org/en/jsr/detail?id=151.

Sutton, R. S., & Barto, A. G. (1998), Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). Cambridge, MA: The MIT Press.

Wang, Z., Zhu, X., & Singhal, S. (2005). Utilization and slo-based control for dynamic sizing of resource partitions. In Proceedings of the 16th IFIP/IEEE Ambient Networks international conference on Distributed Systems: operations and Management (pp133-144). Berlin, Germany: Springer.

Xu, J., Zhao, M., Fortes, J., Carpenter, R., & Yousif, M. (2007). On the use of fuzzy modeling in virtualized data center management. In Proceedings of the International Conference on Autonomic Computing (p 25). Washington, DC: IEEE Computer Society.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., & Currey, J. (2008). DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In Proceedings of the Symposium on Operating System Design and Implementation (pp. 987-994). New York, NY: ACM.

KEY TERMS

Distributed data platforms, self-tuning, elastic scaling, automatic resource provisioning, workload monitor, transactional consistency.

KEY TERMS & DEFINITIONS

Self-tuning system: a system that is able of optimizing its own internal running parameters in order to maximize or minimize the fulfillment of an objective.

Elastic scaling: property of a system that allows its users to use and free resources as needed, while ensuring de that the scale of system is changed in real-time to meet the demands of varying workloads.

Transactional memory: programming paradigm that tries to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way.