Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

D1.2: Enabling Technologies Report

Date of preparation: 30 November 2010
Start date of project: 1 June 2010
Duration: 36 Months

Contributors

Emmanuel Bernard, Red Hat
Joao Cachopo,INESC-ID
Nuno Carvalho,INESC-ID
Jonathan Halliday,Red Hat
Mark Little,Red Hat
Jorge Martins,INESC-ID
Bob McWhirter,Red Hat
Francesco Quaglia,CINI
Joao Nuno Silva,INESC-ID
Paolo Romano,INESC-ID
Vittorio Ziparo,Algorithmica

Table of Contents

1 Introduction

This documents surveys candidate technologies for meeting the user requirements of the Cloud-TM project. The list reflects the outcome of already carried out work within the project. It could therefore be non-exhaustive of the complete set of technologies the will be exploited while finalizing the project, as additional technologies could be identified in order to tackle issues that will potentially arise during the execution of the project.

The Cloud-TM project aims at designing, building, and evaluating the Cloud-TM platform, which is a middleware for service implementation in Cloud Computing infrastructures, addressing the following major challenges:

- offering a simple and intuitive programming model for the implementation of services deployed in Cloud computing platforms. Specifically, by letting service developers focus on delivering differentiating business value, instead of managing low-level, error prone mechanisms (such as inter-process synchronization, data distribution, persistence and fault-tolerance), Cloud-TM will allow a major reduction of the costs associated with the development process.
- minimizing the monitoring and administration costs/efforts by automating the provisioning of resources from the Cloud, with the twofold objective of meeting user-specified Quality of Service and allowing operational costs to be consistent with predetermined budgets.
- maximizing the scalability and efficiency of the user-level services (thus improving the cost/benefit ratio in the Cloud Computing usage-based pricing model) via autonomic paradigms providing self-tuning capabilities aimed at optimizing the platform's performance in face of fluctuations of the number of allocated resources and of the workload characteristics.

On the basis of the above challenges, categories of technologies relevant for this document include (I) software transactional memory layers, either centralized or distributed, and related mechanisms, (ii) frameworks for distributed computing and (iii) tools for Cloud Computing.

1.1 Relationship with other deliverables

The technologies described in this document will serve as a reference for the majority of the future technical deliverables of the project. Being a complement of the deliverable D1.1 “User requirements report”, this document will strongly influence the definition of deliverable D2.1 “Architecture Specification Draft” and D2.2 “Preliminary prototype of the RDSTM and RSS”, which will be developed during the next 6 months.

2 Software Transactional Memory (STM) Layers

The proliferation of shared-memory multicore and multiprocessor systems has substantially increased the opportunity to exploit parallelism. However, in shared memory systems, parallel task-execution requires synchronization mechanisms for accessing shared data. Traditional concurrent programming models usually rely on locking mechanisms to guarantee isolation: when one or more locks on some shared data have been acquired by a thread/process, than no other threads/process can access them. Anyway, lock-based programming is not a simple task. Generally, coarse-grained locking provides low performance and applications do not scale well. On the other hand, fine-grained locking improves performance and scalability, but it's more complex and error prone. E.g., deadlocks, livelock, priority inversion and convoying are more likely to to become manifest with fine-grained locking. Also, according to classical programming models, programmers are in charge of explicitly handling these problems.

Software Transactional memories (STM) are a promising mechanism for tackling the difficulties associated with parallel programming in shared memory systems. STMs provide a simple programming model based on the concept of transaction: the programmer identifies a group of operations to be executed atomically and in isolation from operations by other threads/processes, and tags this group with a begin statement and an end statement. The underlying STM layer ensures atomic and isolated execution of such block of code (or transaction). Given that the concurrency management mechanisms provided by STM layers deliver performance comparable with fine-grained locking, the STM approach can be employed to substantially simplify programming by completely masking synchronization issues) without sacrifying performance,

The design space of an STM is quite wide so that different STM layers can provide differentiated guarantees. For instance, STM implementations can provide different progress guarantees, ranging from deadlock-freedom, to obstruction-freedom or lock-freedom [38]. Another key design criteria is related to the kind of atomicity guarantee ensured in the case of concurrent execution of transactional and non-transactional code [38]. Also, STMs can be word-based, where applications access data words in memory, or object-based, where data are accessed as objects. Very recent STM proposals also ensure opacity [41], namely they ensure that transactions always read consistent data, independently of whether they are eventually aborted or committed.

In the following, we will overview some representative STM layers.

2.1 DSTM2

DSTM2 (second release of the project DSTM) [37] is a flexible framework for implementing object-

based software transactional memories. It is a software transactional memory library implemented as a collection of Java packages supporting a transactional API.

A key aspect of DSTM2 is that it provides the users with the ability to "plug in" their own synchronization and recovery mechanisms in the form of transactional factories that transform stylized sequential interfaces into transactionally synchronized data structures. Like its predecessor, DSTM2 also allows users to customize their techniques for managing contention.

The DSTM2 library assumes that multiple concurrent threads share data objects, and provides a new kind of thread that can execute transactions, which access shared atomic objects. DSTM2 threads provide methods for creating new atomic classes and executing transactions.

The library provides a novel way to define atomic classes. In fact, for each atomic class, the programmer defines a stylized sequential interface, which is a named collection of method signatures satisfying certain simple consistency conditions. The programmer then passes this interface to the constructor of a transactional factory, which uses a combination of reflection and run-time class synthesis to create an anonymous class implementing the original interface, whose methods are transactionally synchronized. The constructed factory creates instances of this anonymous class. Reflection and run-time class synthesis do not occur each time an atomic object is created using a factory. They occur once per atomic class, in particular, when the factory is constructed.

Transactional factories embody different transactional synchronization and recovery techniques. In the library there are two examples of factory classes which employ very different techniques (e.g., one is nonblocking, the other uses locks).

DSTM2 allows the programmers to also define their own factories since the interface is passed to the transactional factory constructor. After that, the choice of how to implement atomic objects is a run-time decision, which can provide supports for experimentation with different synchronization and recovery techniques.

DSTM2 supports simple atomic classes that provide transactional get and set methods for "virtual fields" but it also provides a straightforward manner to implement more complex data structures.

DSTM2 also provides the ability to register methods (in the form of Callable<Boolean> objects) to be called when important transactional events occur. For example, some programmers can register a method that can veto a transaction’s attempt to commit (the so-called validation step). Some other programmers can also register methods to be run immediately after a transaction commits or aborts (useful for cleaning up data structures). The latter service could be considered as essential for giving factories the flexibility needed to implement a range of strategies.

Finally, DSTM2 provides supports for user-defined contention managers.

2.2 TL2

Transactional Locking II (TL2) [39] is an STM based on its precursor TL algorithm [40]. It uses the commit-time locking (CTL) algorithm together with a global version-clock technique. TL2 provides weak atomicity and user code is guaranteed to operate only on consistent memory states. Further it works with an open memory system. A TL2 release was provided for x86 architecture and it was written in C.

CTL does not acquire locks upon accessing data items. Instead, lock acquisition is delayed to commit time and only involves written data items (write-locks). This choice enhances concurrency with respect to conventional lock-based schemes by, e.g., avoiding to block transactions issuing a write operation on a data item that has already been read/written by a concurrent transaction. The drawback is that a late detection of conflicts (i.e. at commit time) can cause to waste a large amount of work. Given the absence of read-locks, consistency is ensured via a validation mechanism used to notify transaction *T*, which speculatively reads a data item *x*, about the fact that *x* was overwritten by some concurrent transaction *T'* preceding *T* in the commit order. To this end, CTL employs a versioning scheme that associates a timestamp value with each data item, referred to as Write-Version-Clock (WVC). The generation of WVC values relies on a unique Global-Version-Clock (GVC), which is read by any transaction upon startup, and is atomically increased upon transaction commit. The updated value is used as the new WVC value for all the data items written by the committing transaction. When validating a transaction against a read data item *x*, two actions are performed:

1. it is checked whether there is a write-lock being held on *x* (which implies that another transaction has written *x* and is currently within its commit phase);
2. it is checked whether the current timestamp associated with *x* is grater than the timestamp read by the transaction upon starting up (which indicates that some concurrent transaction has overwritten *x* and has already been committed).

If one of the previous checks fails, the transaction gets aborted. This validation scheme is used upon each read operation and also at commit time. As far as write operations are concerned, in CTL they are buffered within a private workspace until the commit phase. When a transaction attempts to commit, it first acquires the write-locks for all the data items within its write-set. If any of these lock acquisitions fails (due to lock holding by some other transaction), the transaction is aborted. Otherwise, the transaction increments the GVC and tries to validate all the data items it has within its read-set (according to the aforementioned validation procedure). If the validation fails

for at least one item within the read set, the transaction gets aborted. If no abort occurs, the data within the write-set are copied back to their original memory locations, updating their WVCs with the value of the GVC. All the acquired locks are released at the end of the commit phase, or upon the abort. Execution of read-only transaction is simpler. A transaction read GVC upon startup it is validated only upon execution of a read operation. If none of these validations fails than the transaction successfully commits, otherwise it aborts.

2.3 JVSTM

JVSTM [42] implements a multi-version scheme which is based on the abstraction of a *versioned box* (VBox). A VBox is a container that keeps a tagged sequence of values - the history of the versioned box. Each of the history's values corresponds to a change made to the box by a successfully committed transaction and is tagged with the timestamp of the corresponding transaction. To this end, JVSTM maintains an integer timestamp, *commitTimestamp*, which is incremented whenever a transaction commits. Each transaction stores its timestamp in a local *snapshotID* variable, which is initialized at the time of the transaction activation with the current value of *commitTimestamp*. This information is used both during transaction execution, to identify the appropriate values to be read from the VBoxes, and, at commit time, during the validation phase, to determine the set of concurrent transactions to check against possible conflicts. JVSTM relies on an optimistic approach which buffers transactions' writes and detects conflicts only at commit time, by checking whether any of the VBoxes read by a committing transaction T was updated by some other transaction T' with a larger timestamp value. In this case T is aborted. Otherwise, T commitTimestamp is increased, its snapshotID is set to the new value of commitTimestamp and the new values of all the VBoxes it wrote are atomically updated.

JVSTM is the concurrency control engine of the FenixFramework, a distributed and persistent STM that is currently used in a production environment. FenixFramework will be described in Section 3 of this document.

3 Distributed Software Transactional Memories

Distributed Software Transactional Memories represent the natural evolution of STMs, providing developers with a single-system image abstraction that shelters them from both the issue of concurrency (as in traditional STMs) and distribution.

This is a very recent area of research and only a handful of solutions have been proposed so far. Specifically, the only distributed (not fully replicated) STM solutions we are aware of are DMV, DiSTM, Cluster-STM and the Fenix Framework. In this section, we will overview the first three DSTMs and we will describe Fenix Framework in more details, since this is the only solution we are aware of that takes into account issues related to data persistence and fault-tolerance, which represent key requirements for the Cloud-TM platform.

3.1 DMV

Distributed Multi-Versioning (DMV) [43] exploits the simultaneous presence of different versions of the same transactional dataset across the replicas, to implement a distributed multi-versioning scheme (DMV). Like centralized multi-version concurrency control schemes (e.g. JVSTM), DMV allows read-only transactions to be executed in parallel with conflicting updating transactions. This is achieved by ensuring that the former is able to access older, committed snapshots of the dataset. However, in DMV each replica maintains only a single version of each data granule, and explicitly delays applying (local or remote) updates to increase the chance of not having to invalidate the snapshot of currently active read-only transactions (and to consequently abort them). This allows DMV to avoid maintaining multiple versions of the same data at each node, unlike in conventional multi-version concurrency control solutions (although DMV requires buffering the updates of not yet applied transactions). On the other hand, while multi-version concurrency control solutions provide deterministic guarantees on the absence of aborts for read-only transactions, the effectiveness of the DMV scheme depends on the timing of the concurrent accesses to data by conflicting transactions.

3.2 Cluster-STM

Cluster-STM [44] focuses on the problem of how to partition the data-set across the nodes of a large scale distributed Software Transactional Memory. This is achieved by assigning to each data item a home node, which is responsible for maintaining the authoritative version (and the associated metadata) of the data item. The home node is also in charge of synchronizing the accesses of conflicting remote transactions. In Cluster-STM any caching or replication scheme is

totally delegated to the application level, which has then to take explicitly into account the issues related to data fetching and distribution, with an obvious increase in the complexity of the application development. Further, Cluster-STM treats the processors as a flat set, not distinguishing between processors within a node and processors across nodes, and not exploiting the availability of shared memory between multiple cores/processors on each replica to speed up intra-node communication. Finally, Cluster-STM is constrained to run only a single thread for each processor.

3.3 DiSTM

Distributed STM (DiSTM) [45] does not rely on multi-versioning schemes, but, analogously to DMV, relies on a distributed mutual exclusion mechanism scheme. Mutual exclusion is aimed at ensuring that at any time there are no two replicas attempting to simultaneously commit conflicting transactions. The use of multiple leases, based on the actual datasets accessed by transactions, permits to partially alleviate the performance problems incurred by the serialization of the whole (distributed) commit phase. However, this phase may still become a bottleneck with conflict intensive workloads. Additionally, in DiSTM the lease establishment mechanism is coordinated by a single, centralized, node which is likely to become a performance bottleneck for the whole system as the number of replicas increase. This work is also a framework that was designed for prototyping STM coherence protocols, but this module is the only configurable part of this framework.

3.4 Fenix Framework

The Fenix Framework [46] germinated in the Centro de Informática do Instituto Superior Técnico to provide an integrated academic management system for Instituto Superior Técnico (IST). An open-source philosophy and the involvement of teachers, students and professionals has resulted in an exemplary environment of development and learning. The first and foremost application to be developed was the FenixEDU application, a web application that supports a wide range of academic activities in the IST campus (management of web pages for different courses, student enrollment, etc) and is used today in a few higher education institutions.

In its first version, the Fenix framework adopted a classic three-tiered architecture, with the application logic implemented in Java and its data stored in a relational DBMS. At this stage, Fenix Framework relied on an Object/Relational Mapping (ORM) tool to persist the domain objects in the database and address the issues associated with the mismatch between the object oriented model, used by programmers to develop the application’s business logic, and the relational data

model embraced by classic DBMSs. In addition, in order to reduce the frequency of access to the back-end database, the Fenix Framework relied on a local cache at the middle tier.

To control the concurrent access to the domain entities, the Fenix Framework relied on explicit lock-based interfaces to synchronize read and write operations [7]. Unfortunately, this lock-based approach to concurrency was highly error-prone, as programmers often forgot or misplaced the acquisition of locks, causing frequent consistency problems into the domain data. Moreover, with the increased usage of the system, also the first performance problems appeared. After some performance profiling, these were attributed to the overheads incurred in the acquisition and in the management of locks by the operations that accessed many thousands of objects.

The first, crucial step taken by the FenixFramework to address these issues was the introduction of a Software Transactional Memory [32] layer implementing the abstraction of atomic, isolated transaction for the in-memory objects maintained at the middle-tier.

Specifically, the Fenix Framework was augmented with JVSTM, an STM developed as a JAVA library that relies on a software based implementation of a multi-version concurrency control scheme [1], providing excellent performance for read-only transactions (largely predominating in reference benchmarks for Web applications [35, 36], as well as in the FenixEDU's workload), because they incur in negligible book-keeping overheads and are sheltered from the possibility of blocks or aborts. The integration of JVSTM within the architecture of the Fenix Framework was designed so to achieve total transparency from the developer's perspective, and provided benefits not only in terms of performance (thanks to the elimination of the overheads associated with lock acquisition and management), but also in terms of robustness (thanks to the avoidance of the error-prone manual management of locks) and simplification of the programming model (quantifiable in terms of reduction of lines of code to be developed and debugged [3]).

Serving a population of 12000 students, 900 faculty and 800 administrative members and faced with a steadily increasing traffic volume, the FenixEDU application was soon forced to address the problem of scaling out the TM-enabled application server. As a first step in this direction, a very simple replica synchronization scheme orchestrated by a centralized back-end database is currently being employed. Figure 1 depicts the architecture of the current version of the Fenix Framework, which is being used in the production environment of the FenixEDU application. At the first tier, it uses a load balancer to distribute the client requests among several application servers. The application servers rely on a (logically) centralized DBMS to store the data and maintain the cache of the servers consistent using the following mechanism.

Essentially, each application server is required to access the database every time it starts a new transaction (whether read-only or not) to check whether its local cache is still up-to-date. The

detection of any conflict developed during transactions' execution is performed at commit time via a sequential validation phase performed by checking whether the readset of the committing transaction T intersects with the writesets (stored by the database) of any transaction that has committed before T. Unfortunately, even though this approach is very simple, the reliance of the current replication solution on an external data storage causes large performance overheads, strongly limits concurrency, and suffers of a single point of failure.

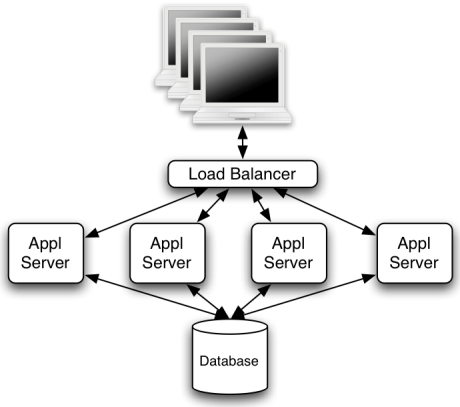


Figure 1: The current Fenix Framework architecture.

3.4.1 Decoupling Replica Synchronization and Data Persistence

As discussed in the previous section, in its current version, the Fenix Framework suffers of several scalability issues imputable to its reliance on an external DBMS to ensure the consistency of the distributed JVSTM instances hosted by each application server.

To address this issue, in [4] we have recently proposed a new architecture that neatly decouples the issues related with the synchronization of the replicated TM layer with those concerning the persistence of data. In the envisioned architecture, which is depicted also in Figure 2, the consistency of the TM-enabled application server replicas is no longer dependent on a centralized DBMS. Conversely, the application server replicas coordinate the execution of transactions by directly communicating among them, leveraging on the services provided by a Group Communication System to propagate the updates and reach cluster-wide agreement on the outcome (commit/abort) of transactions.

Coping with the issues related to the concurrent execution of distributed transactions directly at the application server replicas' level, rather than relying on the assistance of a standard relational DBMS, provides two main advantages.

First, the achievement of a neater separation of logical concerns: once the TM tier has autonomously ensured the consistency of a transaction, the back-end just has to be able to atomically persist the corresponding updates. This permits to rely on much simpler, and more lightweight, persistence solutions than fully-fledged relational databases (which incur in the unnecessary overheads associated with SQL or complex concurrency control mechanisms [34]).

Further, the reliance on a standard, relational DBMS as the coordinator of the TM replication protocol forces to implement the whole replication protocol by exploiting exclusively standard SQL interface. Being SQL designed for other purposes, it can significantly hinder the development of even basic mechanisms typically employed by any transactional replication scheme (e.g., synchronous propagation of state updates to other replicas).

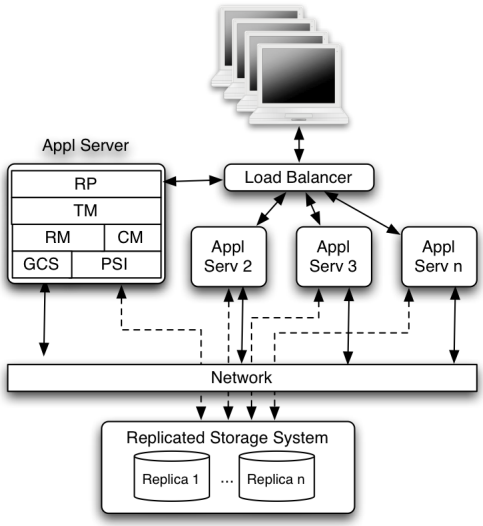


Figure 2: The proposed DTM based architecture.

Let us now analyze more in detail the architecture illustrated in Figure 2. The incoming requests are dispatched by a load-balancer to a set of replicated application servers, which rely on a replicated persistent storage for ensuring durability. Note that the latter is depicted as a logical independent component, even though it could be physically collocated in the same machines also hosting the application server to enhance locality between the application logic and the (persistent)

data.

Each application server hosts the following components: a Request Processor (RP), responsible for receiving the requests and activating the transactional logic; a TM instance, extended with a reflective interface that externalizes key information about the transactions' execution state (e.g., transactions' readsets and writesets), which are normally encapsulated by existing TM implementations; a Cache Manager (CM), responsible for implementing caching policies (e.g., prefetching, eviction strategies) based on the application access patterns; a Replication Manager (RM), implementing the distributed coordination protocol required for ensuring replica consistency (an overview of the TM replication protocols currently under development/evaluation will be provided in the remainder of this paper); a Persistent Store Interface (PSI), providing APIs to interact with a replicated storage system; a Group Communication Service (GCS), responsible for maintaining up-to-date information regarding the membership of the group of application servers (including failure detection) and providing the required communication support for the coordination among the servers.

3.4.2 The BFC Protocol

The Replication Manager, being in charge of ensuring the consistency of the DTM layer, represents a fundamental building block of the architecture described in Section 1.2. In this section we overview one of our recent results concerning the design and evaluation of TM replication mechanisms, namely the Bloom Filter Certification (BFC) protocol [8].

The replica synchronization scheme employed in D2STM is inspired by recent database replication approaches [25], where replica consistency is achieved through a distributed certification procedure which, in turn, leverages on the properties of an Atomic Broadcast (Abcast) [10] primitive. Unlike classic eager replication schemes (based on fine-grained distributed locking and atomic commit), that suffer of large communication overheads and fall prey of distributed deadlocks [13], certification based schemes avoid any onerous replica coordination during the execution phase, running transactions locally in an optimistic fashion. The consistency of replicas (typically, 1-Copy serializability) is ensured at commit-time, via a distributed certification phase that uses a single Atomic Broadcast to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of (a minority of) replica failures. Furthermore, unlike classic read-one/write-all approaches that require the full execution of update transactions at all replicas [1], only one replica executes update transactions, whereas the remaining replicas are only required to validate the transaction and to apply the resulting updates. This allows to achieve high scalability levels even in the presence of write-dominated workloads, as long as the transaction conflict rate remains moderate [25].

However, the efficiency of any transactional replication scheme is much more strained in TMs than in databases, given that the average transaction's execution time in TMs is typically several orders of magnitude smaller than in databases (in [28], for instance, it was shown that 50% of write transactions complete in less than 200μsecs when considering standard TM benchmarks). This translates into a corresponding increase of the overhead associated with the inter-replica coordination activities, urging for novel solutions aimed at minimizing such costs.

The BFC protocol aims at achieving exactly this goal, requiring just a single ABcast to commit a transaction and avoiding to flood the network with large messages carrying the whole transaction's read set. The latter aspect is particularly important given that it is well-known that the ABcast latency is significantly affected by the size of transmitted messages [15, 20] and that the transactions' read sets are frequently very large in web applications.

BFC achieves such a result by exploiting the space-efficient encoding properties of Bloom Filters (BF), whose fundamentals we briefly recall in the following for the sake of clarity (the interested reader may refer to [2] for a recent survey on BF and on their applications). BFs are data structures that permit to test whether an element is a member of a set, avoiding the encoding of the whole set, but rather permitting to store a much more compact representation of it. This comes, however, at the cost of incurring in false positives (i.e., an element may appear to be present in the set, whereas it is not), albeit false negatives are, on the other hand, not possible. More in detail, a Bloom Filter representing a set of n elements from a universe U consists of an array of m bits, initially all set to 0 (zero). The filter uses k independent hash functions with range $\{1,\ldots,m\}$, which map each element in the universe to a random number uniformly over the range. To add an element $x \in S$ to a BF, x is fed to each of the k hash functions. The array positions output by the k hash functions are then all set to 1. To determine whether an item y belongs in S, the values of the bits are checked. If even only one of these bits is 0, it means that y is not a member of S (with no possibility of mistakes). If all are set to 1, then x may be in S, although this may be wrong with some probability, called *false positive* probability. Interestingly, the probability of a false positive f for a single query to a Bloom Filter can be known beforehand, once the number of bits used per item m/n and the number of hash functions k are fixed, by using the following formula:

$$f=(1-e^{-kn/m})^k$$

We can now start describing the BFC scheme. Similarly to existing certification-based transactional replication schemes, in BFC incoming transactions are locally processed in an optimistic fashion, avoiding any inter-replica synchronization scheme during transaction execution. Further, by

leveraging on the JVSTM multi-version scheme, the BFC ensures that read-only transactions are always provided with a consistent committed snapshot. This spares them from the risk of aborts and permits to obviate the need for replica coordinations even during the commit phase. Overall, the overhead incurred in by read-only transactions due to the replication scheme is in practice almost nullified.

For what concerns update transactions, at commit time these are first locally validated to detect any local conflicts. If the local validation phase is successfully passed, the Replication Manager encodes the transaction's read set (i.e., the set of identifiers of all the objects read by the transaction) in a BF, and ABcasts it along with the transaction write set.

As in classical non-voting certification protocols, e.g. [25], update transactions are validated by all replicas once they are ABcast-delivered. At this stage, replicas check whether the BloomFilter of the validating transaction contains any item updated by any concurrent transactions. If no match is found, given that a BF provides no false negatives, then the transaction may be safely committed. Otherwise the transaction is aborted.

Given that the occurrence of false positives leads to the generation of unnecessary aborts, the BF size is set so to ensure that the probability for a false positive to induce a transaction abort is bounded by a user-tunable threshold, which we denote as *maxAbortRate*. The problem here is that *maxAbortRate* is a function of the number of queries *q* that will be performed on the BF during the transaction's validation phase, but the BF has to be constructed when the transaction enters the commit phase. At this time, however, it is not possible to predict the value of *q*, which is determined by the number of transactions that will commit before the transaction is ABcast-delivered and by the size of the write sets of each of these transactions. Neither of these are known when the BF is created. On the other hand, it is important to highlight that any error in estimating *q* does not compromise consistency, but may only lead to deviations from the target *maxAbortRate* threshold. To tackle this problem, BFC uses a lightweight heuristic that estimates *q* through the moving average across the number of BF queries performed during the validation phase of the last *c* transactions to have been ABcast-delivered. Once *q* is estimated, we can then determine the number *m* of bits in the BF by considering that the false positives for any distinct query are independent and identically distributed events which generate a Bernoullian process where the probability of occurrence of a single event (namely, a false positive during a single query) is given by Equation 1. After some simple maths, we obtain the following expression for the BF's size:

$$m = \left\lceil -n \frac{log_2(1-(1-maxAbortRate)^{\frac{1}{q}})}{ln\ 2} \right\rceil$$

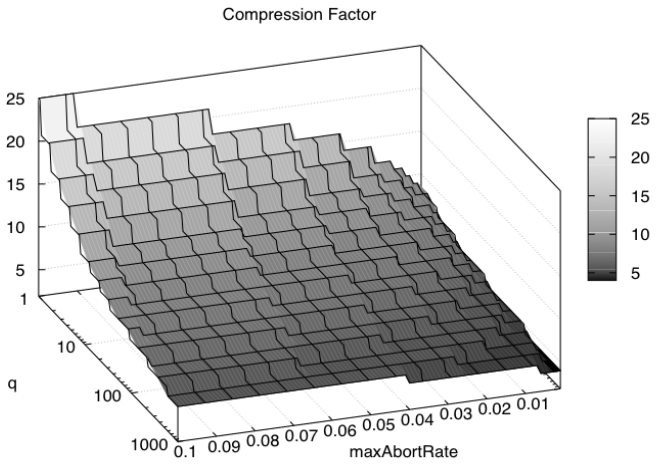


Figure 3: Compression factor achieved by BFC considering the ISO/IEC 11578:1996 UUID encoding.

The striking reduction of the amount of information exchanged, achievable by the BFC scheme, is clearly highlighted by the graph in Figure 3, which shows the BFC's compression factor (defined as the ratio between the number of bits for encoding a transaction's readset with the ISO/IEC 11578:1996 standard UUID encoding, and with BFC) as a function of the target *maxAbortRate* parameter and of the number *q* of queries performed during the validation phase. The plotted data shows that, even for marginal increases of the transaction abort probability in the range of [1%-2%], BFC achieves a [5x-12x] compression factor, and that the compression factor extends up to 25x in the case of 10% probability of transaction aborts induced by a false positive of the Bloom Filter.

4 Related Java 2 Enterprise Edition Technologies

The JavaTM 2 Platform, Enterprise Edition (J2EE), was released by Sun in the late 1990s and defines the standard for developing multi-tier enterprise applications. J2EE simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically, without complex programming.

A typical J2EE Enterprise Application provides its Internet interfaces via web pages, e-mail and Web services. As for Intranet interfaces, J2EE supports both remote procedure call and message oriented integration with legacy applications and supports interaction with SQL based relational databases.

The J2EE standard represents collaboration between leaders from throughout the enterprise software arena - OS and database management system providers, middleware and tool vendors and component developers. It defines a robust, flexible platform that can be implemented on the wide variety of existing enterprise systems currently available, and that supports the range of applications that IT organizations need to keep their enterprises competitive.

In the following we are going to overview some of the components/projects of the J2EE technology that will have stronger influence on the development of the Cloud-TM platform.

4.1 JBossAS

JBoss Application Server is the most widely used Java application server on the market. Hundreds of professional open source developers have contributed to the JBoss Application Server over the years and community contributors are not only welcome but encouraged. In fact all JBoss employed contributors to the JBoss Application Server were hired from the community and each of them contributed to an open source project in one way or another.

As Java EE certified platform for developing and deploying enterprise Java applications, Web applications, and Portals, JBoss Application Server provides the full range of Java EE 5 features as well as extended enterprise services including clustering, caching, and persistence.

4.2 JBossTS

JBoss Transactions (JBossTS) is an open source transaction manager that can be deployed within a range of application servers, containers or run stand-alone. Over the past 20 years it has been used extensively within industry and to drive standards including the OMG and Web Services.

JBossTS is a middleware solution that supports mission-critical applications in distributed computing environments. It plays a critical role in building reliable, sophisticated e-business applications guaranteeing absolute completion and accuracy of business processes. JBossTS supports "multimodal transaction processing" by enabling reliable transactions to span from front-end e-commerce applications to back office systems and beyond the enterprise firewall to business partners - across any system, anywhere in the world.

For the majority of its life JBossTS (aka Arjuna transaction system) has been driven by standards and has driven standards. As such, it supports all of the relevant enterprise transaction specifications and standards including JTA [47], and Web Services transactions. However, with the advent of multi-core technologies and ubiquitous computing, there has been a shift away from some of these specifications to more fine grained and in-memory transactional solutions within Java Enterprise Edition frameworks. As such, JBossTS has become the basis for several STM implementations which are being developed as projects within JBoss for inclusion within products over the next few years.

4.3 Data Services Platform

The Data Services Platform [48] is a JBoss project that aims at using Hibernate to persist data in a grid (key/value specifically) and still let users use the familiar JPA API. Another aspect of this project is to explore the structure used to persist an object model into a grid keeping as much of the relational model benefits as possible.

Specifically, the Data Services Platform will be composed of three main sub-components: Hibernate Core, Hibernate OGM and Hibernate Search.

4.3.1 Hibernate Core

Hibernate Core [49] provides persistence for idiomatic Java, dealing with the issue of how to persist an Object-Oriented domain model into relational databases (RDBMS). In the world of Object-Oriented applications, there is often a discussion about using an object database (ODBMS) as opposed to a RDBMS. We are not going to explore that discussion here. Suffice it to say that RDBMS remain a very popular persistence mechanism and will so for the foreseeable future.

The *Object-Relational Impedance Mismatch* (sometimes called the *paradigm mismatch*) happens due to the discrepancies between the object-oriented and the relational models. RDBMSs represent data in a tabular format, whereas object-oriented languages, such as Java, represent it as an interconnected graph of objects. Loading and storing graphs of objects using a tabular relational database exposes us to several mismatch problems, namely:

1. Granularity: Sometimes you will have an object model which has more classes than the number of corresponding tables in the database (the object model is more granular than the relational model);
2. Subtypes (inheritance): Inheritance is a natural paradigm in object-oriented programming languages. However, RDBMSs do not define anything similar on the whole;
3. Identity: A RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).

Hibernate has the following advantages:

- Natural Programming Model: Hibernate lets you develop persistent classes following natural Object-oriented idioms including inheritance, polymorphism, association, composition, and the Java collections framework.
- Transparent Persistence: Hibernate requires no interfaces or base classes for persistent classes and enables any class or data structure to be persistent. Furthermore, Hibernate enables faster build procedures since it does not introduce build-time source or byte code generation or processing.
- High Performance: Hibernate supports lazy initialization, many fetching strategies, and optimistic locking with automatic versioning and time stamping. Hibernate requires no special database tables or fields and generates much of the SQL at system initialization time instead of runtime. Hibernate consistently offers superior performance over straight JDBC coding.
- Reliability and Scalability: Hibernate is well known for its excellent stability and quality, proven by the acceptance and use by tens of thousands of Java developers. Hibernate was designed to work in an application server cluster and deliver a highly scalable architecture.
- Extensibility: Hibernate is highly customizable and extensible.
- Comprehensive Query Facilities: Including support for Hibernate Query Language (HQL), Java Persistence Query Language (JPAQL), Criteria queries, and "native SQL" queries; all of which can be scrolled and paginated to suit your exact performance needs.
- Direct use of the object (de)hydration.
- Direct use of the Java Persitence API (JPA).

4.3.2 Hibernate OGM

Hibernate Object Grid Mapper aims at providing the same benefits enjoyed by Hibernate Core, but persist the data in a Grid, namely in a key value store such as Infinispan (see Section 5.3.1). JPA has become the *de facto* standard for Java developers dealing with object persistence. Both its simplicity and powerful query language let developers focus on their business application rather than low level boiler plate code.

Hibernate OGM is composed of several components:

1. A CRUD (Create Read Update Delete) engine providing the full JPA2 semantic including cascading, transparent state propagation, lazy loading, polymorphism and more. In particular, support for associations (one to one, one to many, many to one and many to many) is a priority.
2. A search engine capable of executing most JP-QL queries on the data stored in the data grid. The goal here is to offer most of the flexibility expected from Java developers.

Part 1 is provided as a module of Hibernate Core that replaces the low level persistence logic and make use of the data Grid as a persistence store. The underlying persistence format tries to be as close as possible to the foundational concepts of the relational model. The idea being that we can benefit from a couple of key features of the Relational model including abstraction from the moving application model and logic.

4.3.3 Hibernate Search and Teiid

The Hibernate Search project aims at providing full-text search capability to objects stored in a backend. Hibernate Search will be layered on top of either Hibernate Core or Hibernate OGM, thus allowing to target a wide range of backends including key value stores such as Infinispan as well as more traditional RDBMS.

Full text search engines like Apache Lucene [62] are very powerful technologies to add efficient free text search capabilities to applications. However, Lucene suffers several mismatches when dealing with object domain model. Amongst other things indexes have to be kept up to date and mismatches between index structure and domain model as well as query mismatches have to be avoided.

Hibernate Search addresses these shortcomings - it indexes your domain model with the help of a few annotations, takes care of backend/index synchronization and brings back regular managed objects from free text queries. To achieve this, Hibernate Search combines the power of Hibernate and Apache Lucene. In short, Hibernate Search provides query capability to your persistence

domain model and deals with three important mismatches:

1. Structural mismatch: objects have rich types like Date and Numbers, can be polymorphic and have association with one another. None of these properties are present in the indexing technology. Hibernate Search bridges the gap between the two.
2. Synchronization mismatch: Lucene as a standalone product does not update its indexes when an object is updated in the persistence store. Hibernate Search fixes this shortcoming by listening to every change to the persistence storage and propagates them into the indexes.
3. Retrieval mismatch: retrieving data, say via JPA, is different from retrieving data via Lucene, as domain objects are retrieved, managed and have lazy associations. Hibernate Search bridge this gap by making full-text search available as a fourth approach to query in JPA (JP-QL, SQL, Criteria API and now full-text search). To push that even further, Hibernate Search embraces Hibernate and JPA APIs and reuses the same concepts.

Within Cloud-TM it is our intention to rely on Hibernate Search to provide applications with a search engine that will allow to query the data maintained by the platform.

In order to develop a scalable and distributed search engine on top of Cloud-TM, our plan is to extract and extend the query engine of a popular data virtualization system, called Teiid.

Teiid is comprised of tools, components and services for creating and executing bi-directional data services, allowing applications to use data from multiple, heterogenous data stores in a transparent fashion.

Through abstraction and federation, data is accessed and integrated in real-time across distributed data sources without copying or otherwise moving data from its system of record.

The heart of Teiid, which we plan to integrate in the Cloud-TM platform, is a high-performance query engine that processes relational, XML, XQuery and procedural queries from federated datasources. Features include support for homogenous schemas, heterogenous schemas, transactions, and user defined functions.

Our goal is to make use of Teiid's query distribution and aggregation capabilities to make the Cloud-TM search engine scalable, efficient, fault-tolerant but still covering most of SQL features like joins, aggregation operations and more.

4.4 Messaging and Group Communication Systems

View synchronous group communication [57] is a coordination paradigm that eases the

development of multi-participant applications, ranging from replicated servers, cooperative caches, multi-user cooperative applications, just to name a few. The set of protocols that implement group communication services are typically bundled in a package called a group communication toolkit. After the pioneer work initiated two decades ago with Isis [58], many other toolkits have been developed. Appia, Spread, and JGroups are, among others, some of the group communication toolkits in use today. Therefore, group communication is, today, a mature technology that, when correctly used, greatly eases the development of reliable distributed applications. At the same time, group communication is still a hot research topic, as performance improvements and wider applicability are sought.

A group communication toolkit integrates two complementary services: a membership service and a multicast communication service. Informally, the role of the membership service is to provide, to each participant in a distributed computation, information about who is active (or reachable) and who is failed (or unreachable). Such information is called a view of the group of participants. The multicast service allows a member to send a message to the group of participants with different reliability and ordering properties. Both membership and multicast services need to be integrated, since the reliability guarantees are usually defined in the context of the current group view.

From the description above it is clear that group communication is much more than just “yet another reliable multicast protocol” given that, the added value, is the precise semantics that are enforced among the communication and membership services, namely in the presence of faults. By ensuring that the same messages are delivered to multiple destinations, ordered among themselves and with group membership change notifications, each message may be handled by the application in a predictable and globally consistent context.

Complex distributed state maintenance problems are then greatly simplified by the strong semantics of Group Communication. Group communication is therefore found at the core of multiple widely deployed and used middleware products.

4.4.1 Appia

Appia [22] is a layered communication support framework that was implemented in the University of Lisbon. It is implemented in Java and aims at high flexibility to build communication channels that fit exactly in the user needs. The service offered by a communication channel can be statically configured by an XML file or dynamically assembled by the application at run time. The application can create several channels with different service guarantees and send messages to different channels, depending on the service required by each message. In contrast with traditional layered protocols, components of Appia channels can be shared and thus offer multiple related services.

This makes it easy, for instance, that several channels can be bound to the same instance of group membership.

In Appia services are composed in a stack, aiming to provide a given Service. An instance of a composition is called a *service channel*, and each layer of that service channel corresponds to a *service session*, which maintains its own state. These sessions interact through an event-driven model. Thus, a service implementation corresponds to a collection of event handlers. Also, events can be generated by services or come from other processes through the network.

Although Appia is protocol independent, in the sense that it can be used to compose any protocol as long as it respects the predefined interface, it includes an extensive layer library targeted at view synchronous group communication. Namely, it has protocols that implement virtual synchrony, causal order, and several implementations of total order algorithms. The total order algorithms composed with virtual synchrony provide the Atomic Broadcast abstraction, useful (for instance) to ease the implementation of replication protocols.

4.4.2 RAppia

RAppia [29] is an evolution of the previously described group communication toolkit. RAppia uses the same composition model already used in Appia and also includes two control stacks that support the adaptation of the remaining service stacks: a generic and configurable *context sensor* and a *reconfiguration monitor*. The context sensor is a service that handles the local capture of context information and allows for its dissemination. The reconfiguration monitor defines a control channel through which each node receives reconfiguration commands from an external *adaptation manager*, such as altering service parameters or switching implementations.

The RAppia adaptation support addresses three different aspects [29]: i) the service programming model, ii) adaptation-friendly services, and iii) kernel mechanisms. The programming model employed, based on the exchange of events, defines a set of adaptation related default events. These include events to access context information produced by services, events to place services in a quiescent or normal state, and events to handle state transfer, which are useful when switching service instances. There is also an event that allows service parameters to be updated, such as the timeout value of a failure detector. Other two useful properties enforced by the programming model are the use of protocol type hierarchies and message headers. Type hierarchies provide a way for protocols to be organized and tagged given their provided properties, and allow, for example, to reason about alternative implementations present in the system. By being included in the messages exchanged between peers, message headers also allow the exchange of control information. In RAppia, these headers are implemented as a *pool* from which

services can retrieve or add fields. RAppia also includes two control stacks that support the adaptation of the remaining service stacks: a generic and configurable *context sensor* and a *reconfiguration monitor*. The context sensor is a service that handles the local capture of context information and allows for its dissemination. For dissemination, two main mechanisms are provided, one to query nodes about their state and another to send asynchronous notifications to other nodes (periodically, for example). The reconfiguration monitor defines a control channel through which each node receives reconfiguration commands form an external *adaptation manager*. With respect to kernel mechanisms, RAppia has several features that facilitate the run-time adaptation of protocol stacks. For instance, instead of stacking headers, headers are organized as a pool (similarly to Cactus [19]). This prevents headers that are inserted at the sender for layers that are no longer active at the recipient to interfere with the processing of other layers. Also, events for sessions that are in a quiescent are queued by the runtime system. This allows for adapting or even changing a given layer without stopping the entire protocol stack.

4.4.3 JGroups

JGroups is a toolkit for reliable multicast communication. Note that this doesn't necessarily mean IP Multicast, JGroups can also use transports such as TCP. It can be used to create groups of processes whose members can send messages to each other.

The main features include:

- Group creation and deletion. Group members can be spread across LANs or WANs;
- Joining and leaving of groups;
- Membership detection and notification about joined/left/crashed members;
- Detection and removal of crashed members;
- Sending and receiving of member-to-group messages (point-to-multipoint);
- Sending and receiving of member-to-member messages (point-to-point).

The most powerful feature of JGroups is its flexible protocol stack, which allows developers to adapt it to exactly match their application requirements and network characteristics.The benefit of this is that you only pay for what you use. By mixing and matching protocols, various differing application requirements can be satisfied. JGroups comes with a number of protocols (but anyone can write their own), for example:

- Transport protocols: UDP (IP Multicast), TCP, JMS;
- Fragmentation of large messages;

- Reliable unicast and multicast message transmission, lost messages are retransmitted;
- Failure detection: crashed members are excluded from the membership;
- Ordering protocols: Atomic (all-or-none message delivery), Fifo, Causal, Total Order (sequencer or token based);
- Membership;
- Encryption.

JGroups is one of the premier open source group communications implementations, with wide spread adoption. At this stage it is also a core component in the Infinispan architecture, which is described later in Section 5.3.1.

4.4.4 Spread

Spread is a toolkit implemented by researchers of the Johns Hopkins University. It is based on an overlay network that provides a messaging service resilient to faults across local and wide-area networks. It provides services ranged from reliable message passing to fully ordered messages with delivery guarantees. The Spread system is based on a daemon-client model where generally long-running daemons establish the basic message dissemination network and provide basic membership and ordering services, while user applications linked with a small client library can reside anywhere on the network and will connect to the closest daemon to gain access to the group communication services. There are interfaces for Spread in multiple programming languages, including Java.

4.4.5 Generic Service for Group Communication

Unfortunately, each group communication toolkit has a different interface, which differs from every other interface in subtle syntactic and semantic aspects. This hinders the design, implementation and maintenance of applications using group communication and forces developers to commit beforehand to a single toolkit, thus imposing a significant hurdle to portability. Such commitment is undesirable because group communication toolkits are often optimized for specific execution environments. The ability to replace one toolkit by another has the advantage of allowing the application to use the most appropriate toolkit for each deployment scenario. This also prevents emerging loosely coupled service oriented architectures from taking full benefit of view synchronous group communication. To solve this problem, we defined a generic interface that may be used to wrap multiple toolkits. The interface, called Group Communication Service for Java, or simply jGCS [5], has been designed for the Java programming language, but any similar language

could be used. The definition leverages on several design patterns that have recently become common ground of Object Oriented middleware. The interface specifies not only the API but also the (minimum) semantics that allow application portability.

jGCS owns a number of novel features that makes it quite distinct from previous attempts to define standard group communication interfaces. The proposed interface aggregates the service in several complementary interfaces, namely a configuration interface, a message passing interface, and a set of membership interfaces. The configuration interface specifies several opaque configuration objects that encapsulate specifications of message delivery guarantees. These are to be constructed in an implementation dependent manner to match application requirements and then supplied using some dependency injection technique. The message-passing interface exposes a straightforward interface to sending and receiving byte sequences, although concerned with high throughput, low latency and sustainable concurrency models in large-scale applications. Finally, a set of membership interfaces exposes different membership management concepts as different interfaces, that the application might support or need. jGCS supports the recent research results that improve the performance of group communication systems, namely, semantic annotations and early delivery.

This service was implemented in Java and is hosted at SourceForge.net [6]. More details about the jGCS specification can be found in [5]. jGCS was itself implemented in several group communication toolkits: Appia, JGroups and Spread. All these bindings are open source and also available in [6].

4.4.6 Java Message Service

Java Message Service (JMS) [50] is a Java API for working with Message-Oriented Middleware, allowing applications to produce and consume messages. JMS is implemented in many MOM-products, HornetQ, ActiveMQ, IBM MQSeries, JBossMQ, SonicMQ are best known.

JMS supports both "basic" models of message exchange. In the Point-to-point model, each message is sent to a specific queue, and receivers extract messages from the queue(s) established to hold their messages. In the Publish-and-subscribe model, each message is published to a topic and dispatched to all registered subscribers. There's also a rich and flexible definition of what a message is composed of, and strict rules governing store-and-forward messaging, guaranteed delivery, message acknowledgments, transactions, and recovery from failures.

A JMS application is composed of a JMS provider that is a MOM implementing the JMS API. The client application generally uses JNDI for discovering administrative objects such as queues, topics

and connection factories. The connection factory is the root object from which connections are created. A connection is a virtual connection with the JMS provider and is used to create sessions that are single-threaded contexts for producing and consuming messages. Sessions are used for creating message producers, message consumers, and messages.

A JMS message is composed of three basic parts: headers, properties, and the message payload. The property section contains a set of application-defined name/value pairs that are accessible via an extensive assortment of setters and getters on the Message object. The message body can contain any type of application data and is accessible in a number of forms. There are five main message types:

- BytesMessage: A stream of bytes intended for any arbitrary data, binary or otherwise;
- ObjectMessage: A set of objects accessible by Java programs;
- MapMessage: A set of name/value pairs in a variety of data types;
- StreamMessage: Similar to BytesMessage and ObjectMessage, with the added value that a message may be read from and written to using a familiar file stream metaphor;
- TextMessage: Plain text.

Messages can be consumed in either of two ways:

- Synchronously: A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives, or it can time out if a message does not arrive within a specified time limit.
- Asynchronously: A client can register a message listener with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage method, which acts on the contents of the message.

Acknowledgement options:

- Auto mode: When a session uses auto mode, the messages sent or received from the session are automatically acknowledged. This is the simplest mode and expresses JMS's power by enabling once-and-only-once message delivery semantic.
- Duplicates okay mode: When a session uses duplicates okay mode, the messages sent or received from the session are lazily acknowledged. This means that messages might be delivered more than once. This mode enables at-least-once message delivery semantic.
- Client mode: When a session uses client mode, the messages sent or received from the

session are not acknowledged automatically. The application must acknowledge the message receipt.

Transaction options:

- Transacted session: An application can participate in a transaction by creating a transacted session (or local transaction). The application completely controls messages being produced and consumed within this session by either committing or rolling back the session.
- XA Session: An application can participate in a global transaction by creating an XA session and enlisted its associated XA resource with the transaction manager. Note that this part of JMS is optional but broadly implemented.

5 Cloud Related Technologies

Over the last years cloud computing has emerged as a powerful paradigm for deploying, managing and offering services through a shared infrastructure. The rapidly expanding market of commercial Cloud Computing infrastructures currently offers solutions of different flavours depending on the nature of the resources made available on demand by the Cloud platform, including Infrastructure-As-A-Service (IAAS), Platform-As-A-Service (PAAS) and Software-As-A-Service (SAAS).

In this section we overview some of the key IAAS and PAAS Cloud technologies that are more closely related to the Cloud-TM project. More in detail, the set of technologies described in the following are either expected to be integrated into the Cloud-TM platform, or exploited to simplify the accomplishment of some of the project's goals, or simply so prominent in the Cloud computing arena to influence the design and evaluation of the Cloud-TM's middleware.

We start by overviewing in Section 5.1 some of the most popular toolkits for the management of Cloud Infrastructures. Next, in Section 5.2 we describe two popular frameworks for the monitoring of Cloud (and, more in general, distributed) infrastructures. Finally, Section 5.3 overviews some of the more popular persistent storage solutions for the Cloud.

5.1 Cloud Infrastructure Management Toolkits

5.1.1 Eucalyptus

Eucalyptus [51] (Computing Architecture Linking Your Programs To Useful Systems) provides an open source software infrastructure to implement on-premise clouds. It allows to deploy private and hybrid clouds on commonly available IT infrastructures. It is compatible with several Linux distribution and relies on web services technology. It also supports AWS cloud interface allowing to interact with it through a common programming interface. Basically Eucalyptus enables cloud management allowing users to create, control and destroy virtual machines and storage resources. High-level Eucalyptus' components are hierarchically arranged and interact with one another through interfaces defined by means of a WSDL document. These components are the following:

- management platform: it is placed at top of hierarchy and provides a management interface for the cloud; in particular it allows to manage virtual machines, storage and networks, user groups; further it provides other features such as SLA definition and enforcement and resources monitoring.

- cloud controller: it is responsible for high level control of cloud through exposing and managing of virtualized resources via a standard API (Amazon EC2) and a web-based interface; it is the entry-point for users of cloud;
- cluster controller: each cluster is accessed by means of a cluster controller which is executed on a cluster front-end machine which have a network connection with node running cloud controller; it manages virtual machines execution on cluster and virtual network;
- node controller: each node of a cluster runs a node controller which manages the execution of virtual machines on it, the virtual machines images and the virtual network end-point;
- storage controller: it implements and manages dynamic block devices which can be attached to a virtual machine to be used as persistent storage;
- walrus: is a storage service which allows users to store persistent data, organized as buckets and objects; users can create, delete, list buckets, put, get, delete objects, set access control policies; further it can be used also to store virtual machine images.

The modular design allows to support a variety of infrastructures and topologies. In additional several different hypervisors and virtualization technologies are supported.

5.1.2 Open Nebula

OpenNebula [52] is an open source, virtual infrastructure manager that deploys virtualized services on both a local pool of resources and external IaaS clouds. The modular architecture allows integration with several hypervisors and infrastructure configurations. It is compatible with different virtualization tools (Xen, KVM and VMware) and allows to build Hybrid cloud with Amazon EC2 and other providers through Deltacloud. Basic operations for virtual machines deployment, monitoring and control are provided by the *core* through pluggable *drivers*. This makes OpenNebula not tied to any specific environment and makes possible to provide an uniform management layer regardless of the underlying infrastructure. Creation of virtual machines rely on an image repository. Further it allows virtual network and storage creation and configuration.

The core component further provides a management and monitoring interface for the physical hosts. Dynamic placement of virtual machines is managed by the Capacity Manager. It works on basis of a set of pre-defined policies. OpenNebula overcomes some existing tools for virtual infrastructures management in providing some extra functionality as well as configurable workload and resource-aware placement policies or deployment of multi-tier services consisting of groups of inter-connected virtual machines.

5.1.3 RHQ, Jopr and RHEV-M

The Red Hat RHQ project is a systems management suite that provides extensible and integrated systems management for multiple products and platforms across a set of core features such as:

- Monitoring and graphing of values;
- Alerting on error conditions;
- Remote configuration of managed resources;
- Remote operation execution.

The project is designed with layered modules that provide a flexible architecture for deployment. It delivers a core user interface that provides audited and historical management across an entire enterprise. A Server/Agent architecture provides remote management and plugins implement all specific support for managed products.

RHQ is an open source project licensed under the GPL, with some pieces individually licensed under a dual GPL/LGPL license to facilitate the integration with extended packages such as Jopr (now included in RHQ) and Embedded Jopr.

Jopr, which is part of RHQ, is an enterprise management solution for JBoss middleware projects and other application technologies. This pluggable project provides administration, monitoring, alerting, operational control and configuration in an enterprise setting with fine-grained security and an advanced extension model. Jopr is part of the multi-vendor RHQ management project. It provides support for monitoring base operating system information on six operating systems, as well as management of Apache, JBoss Application Server and other related projects.

Red Hat Enterprise Virtualization Manager for Desktops (RHEV-M) is a feature-rich server virtualization management system that provides advanced capabilities for hosts and guests, including:

- Live migration: Allows virtual machines to be moved from one host to another while running with no affect on performance;
- High availability: Guarantees if a host goes down that critical virtual machines are restarted on another host;
- Maintenance manager: Allows hosts to be upgraded and maintained while virtual machines are running;
- Image manager: Enables thin provisioning of disk storage and use of templates to quickly build new virtual machines from standard configurations;

- System scheduler: Allows the Red Hat Enterprise Virtualization Manager to migrate virtual machines across multiple hosts to balance workloads;
- Power saver: Allows the Red Hat Enterprise Virtualization Manager to consolidate virtual machines on a smaller number of hosts during non-peak hours and power down unneeded hosts;
- Search Driven User Interface;
- Consistent Management Environment for Server and Desktop.

5.1.4 BoxGrinder

The BoxGrinder project [53] provides a set of tools aimed at simplifying the creation of virtual machine images, also known as *appliances*. Appliances are preconfigured disk images with operating system and required software ready to run on a selected virtualization platform to do specific job: for example one can have a database appliance or a load balancer appliance. BoxGrinder supports various virtualization (and Cloud) platforms such like: KVM, Xen, EC2, Vmware.

There are two approaches for appliance creation:

- Offline building;
- Online snapshotting.

BoxGrinder uses offline building technique to create appliances which creates the whole disk image in an offline fashion. This means that produced appliance wasn't launched before first use. BoxGrinder uses appliance definition files to describe appliances we want to build. Appliance definition files are plain text files in YAML format for readability and easy of modifications. Here is an example of an appliance definition file:

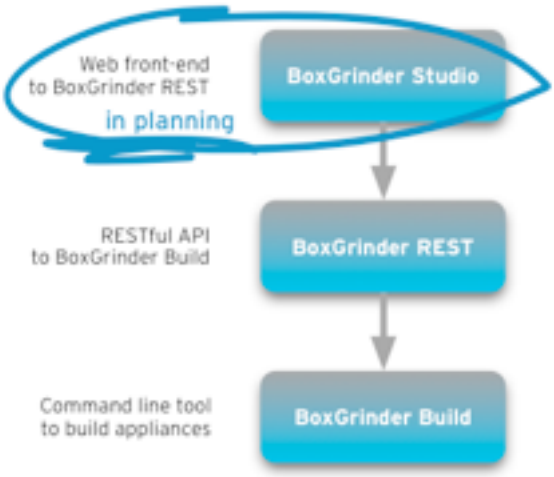
```
name: back-end
version: 1
release: 1
summary: back-end appliance with JBoss AS 6
hardware:
  memory: 512
  partitions:
```

```
    "/":
      size: 2
appliances:
  - fedora-base
packages:
  includes:
    - jboss-as6
    - jboss-as6-cloud-profiles
```

An appliance definition file is composed of several sections, every section being responsible for configuring a different part of the appliance. For example, an appliance info is specified with a name, a version, a release number and a summary. In hardware sections all hardware related parameters can be modified such as memory, cpu count or partition scheme. In the appliance section we can define other appliances. The appliance definition is used to build current appliance and will be the output of all dependent appliance definition files merger. In packages section we specify all software we want to install.

As shown in the picture below, the BoxGrinder project encompasses three main sub-projects:

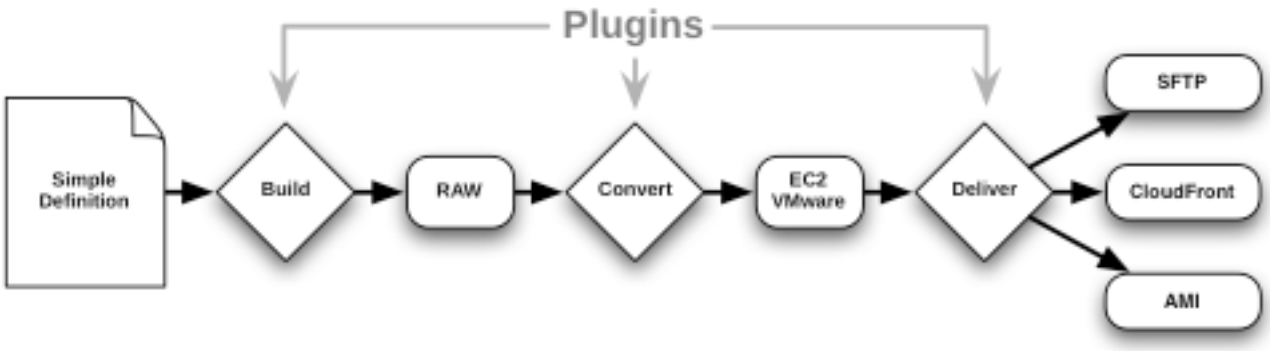
- BoxGrinder Build;
- BoxGrinder REST;
- BoxGrinder Studio.



In the next paragraphs, we will focus on BoxGrinder Build and BoxGrinder REST. BoxGrinder Studio is, in fact, simply a web front-end to BoxGrinder REST and is still currently under-development.

BoxGrinder Build

BoxGrinder Build is a command line tool based on plugin architecture.

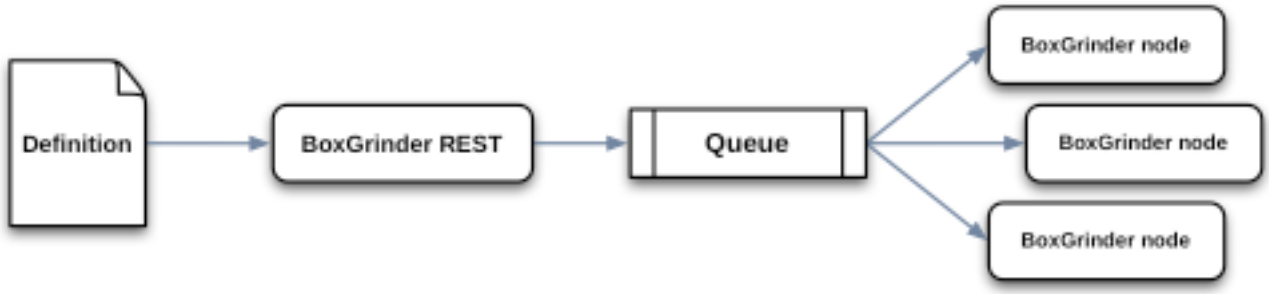


Currently BoxGrinder Build allows for three types of plugins:

- Operating system plugins. The goal of this kind o plugins is to create a base image for selected operating system. The operating systems the are currently supported are Fedora, CentOS and Red Hat Enterprise Linux;
- Platform plugins. A platform plugin converts the images generated from an operating system plugin to a selected platform. A platform could be VMware vSphere or Amazon EC2 for example. Currently supported: VMware, EC2;
- Delivery plugins. A delivery plugin moves the images generated from a platform or operating system plugin to a selected location type. The latter could be a local directory, SFTP server, Amazon CloudFront or a Amazon S3 bucket. Currently supported: Local, SFTP, CloudFront, S3 and EBS.

BoxGrinder REST

BoxGrinder REST is a RESTful interface to a distributed BoxGrinder Build farm.



The BoxGrinder REST server receives an appliance definition file. The definition is parsed and a new build task is created. The task is routed to a build queue. When a build node becomes available it takes a task and executes it. The built appliance is uploaded to a selected repository.

The BoxGrinder REST server manages the nodes. When there are many tasks in the queue it can launch new nodes to distribute the load and therefore speed up the whole system. New nodes are launched in the Cloud via the Deltacloud API. In case there are no new tasks in the queue, BoxGrinder REST can automatically shutdown running builder nodes.

5.2 Monitoring Software

5.2.1 The Lattice Monitoring Framework

The Lattice Monitoring Framework [54] is a management system for service clouds that require a monitoring system that can collect all the relevant data in an effective way. The monitoring system must have a minimal runtime footprint and must not be intrusive, so as not to adversely affect the performance of the network itself or the running service applications. Due to the fact that in a large distributed system there may be hundreds or thousands of measurement probes that can generate data, this framework ensures that the management components only receive data that is of relevance. It would not be effective to have all of these probes sending data all of the time, so a mechanism is needed to control and manage the relevant probes. This monitoring system itself is designed around the concept of producers and consumers. In other words, there are producers of monitoring data, which collect data from probes in the system, and there are consumers of monitoring data, which read it. The producers and the consumers are connected via a network which can distribute the measurements collected. The collection and the distribution of data are decoupled using interfaces in order to change the distribution framework without changing all the producers and consumers. For example, the distribution framework can change over time, say from IP multicast, to an event bus, or a publish/subscribe framework. Making this change should not affect other parts of the system.

In order to increase the power and flexibility of the monitoring, Lattice introduces the concept of a data source. A data source represents an interaction and control point within the system that encapsulates one or more probes. A probe sends a well defined set of attributes and values to the consumers. This can be done by transmitting the data each predefined interval, or transmitting when some change has occurred. The measurement data itself is sent via a distribution framework. These measurements are encoded to be as small as possible in order to maximize the network utilization. Consequently, the measurement meta-data is not transmitted each time, but is kept separately in an information model. This information model can be updated at key points in the lifecycle of a probe and can be accessed as required by consumers.

The goal for the monitoring system is to have fully dynamic data sources, in which each can have multiple probes, with each probe returning its own data. The data sources are able to turn on and turn off probes, or change their sending rate on demand.

In order to distribute the measurements collected by probes within the monitoring system, it is necessary to use a mechanism that fits well into a distributed architecture such as a service cloud. The framework provides a mechanism that allows for multiple submitters and multiple receivers of data without having vast numbers of network connections. For example, having many TCP connections from each producer to all of the consumers of the data for that producer would create a combinatorial explosion of connections. Solutions to this include IP multicast, Event Service Bus, or publish/subscribe mechanism. In each of these, a producer of data only needs to send one copy of a measurement onto the network, and each of the consumers will be able to collect the same packet of data concurrently from the network. Such an approach works well for a service cloud because we have both multiple producers and multiple consumers.

5.2.2 The Ganglia distributed monitoring system

Ganglia [55] is a scalable distributed monitoring system for high performance, distributed computing systems. It is based on a hierarchical design targeted at federations of clusters. It relies on a multicast-based listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state. Within each cluster, Ganglia uses heartbeat messages on a well-known multicast address as the basis for a membership protocol. Membership is maintained by using the reception of a heartbeat as a sign that a node is available and the non-reception of a heartbeat over a small multiple of a periodic announcement interval as a sign that a node is unavailable.

Each node monitors its local resources and sends multicast packets containing monitoring data on a well-known multicast address whenever significant updates occur. Applications may also send on

the same multicast address in order to monitor their own application-specific metrics. Ganglia distinguishes between built-in metrics and application-specific metrics through a field in the multicast monitoring packets being sent. All nodes listen for both types of metrics on the well-known multicast address and collect and maintain monitoring data for all other nodes. Thus, all nodes always have an approximate view of the entire cluster's state and this state is easily reconstructed after a crash.

Ganglia federates multiple clusters together using a tree of point-to-point connections. Each leaf node specifies a node in a specific cluster being federated, while nodes higher up in the tree specify aggregation points. Since each cluster node contains a complete copy of its cluster's monitoring data, each leaf node logically represents a distinct cluster while each non-leaf node logically represents a set of clusters. Aggregation at each point in the tree is done by polling child nodes at periodic intervals. Monitoring data from both leaf nodes and aggregation points is then exported using the same mechanism, namely a TCP connection to the node being polled followed by a read of all its monitoring data.

The implementation consists of two daemons, *gmond* and *gmetad*, a command-line program *gmetric*, and a client side library. The Ganglia monitoring daemon (*gmond*) provides monitoring on a single cluster by implementing the listen/announce protocol and responding to client requests by returning an XML representation of its monitoring data. *gmond* runs on every node of a cluster. The Ganglia Meta Daemon (*gmetad*), on the other hand, provides federation of multiple clusters. A tree of TCP connections between multiple *gmetad* daemons allows monitoring information for multiple clusters to be aggregated. Finally, *gmetric* is a command-line program that applications can use to publish application-specific metrics, while the client side library provides programmatic access to a subset of Ganglia's features.

Monitoring on a single cluster is implemented by the Ganglia monitoring daemon (*gmond*). The *gmond* program is organized as a collection of threads, each assigned a specific task. The collect and publish thread is responsible for collecting local node information, publishing it on a well-known multicast channel, and sending periodic heartbeats. The listening threads are responsible for listening on the multicast channel for monitoring data from other nodes and updating *gmond*'s in-memory storage, a hierarchical hash table of monitoring metrics. Finally, a thread pool of XML export threads are dedicated to accepting and processing client requests for monitoring data. All data stored by *gmond* is soft state and nothing is ever written to disk. This, combined with all nodes multicasting their state, means that a new *gmond* comes into existence simply by listening and announcing.

For speed and low overhead, *gmond* uses efficient data structures designed for speed and high concurrency. All monitoring data collected by *gmond* daemons is stored in a hierarchical hash table

that uses reader-writer locking for fine-grained locking and high concurrency. This concurrency allows the listening threads to simultaneously store incoming data from multiple unique hosts. It also helps resolve competition between the listening threads and the XML export threads for access to host metric records. Monitoring data is received in XDR format and saved in binary form to reduce physical memory usage. In a typical configuration, the number of incoming messages processed on the multicast channel far outweigh the number of requests from clients for XML. Storing the data in a form that is ‘‘closer’’ to the multicast XDR format allows for more rapid processing of the incoming data.

The *gmond* program uses a multicast-based, listen/announce protocol to monitor state within a single cluster. Its main advantages include: automatic discovery of nodes as they are added and removed, no manual configuration of cluster membership lists or topologies, amenability to building systems based entirely on soft-state, and symmetry in that any node knows the entire state of the cluster. Automatic discovery of nodes and eliminating manual configuration is important because it allows *gmond* on all the nodes to be self-configuring, thereby reducing management overhead. Amenability to a soft-state based approach is important because this allows nodes to crash and restart without consequence to *gmond*. Finally, because all nodes contain the entire state of the cluster, any node can be polled to obtain the entire cluster’s state. This is important as it provides redundancy, which is especially important given the frequency of failures in a large distributed system.

Gmond publishes two types of metrics, built-in metrics which capture node state and user-defined metrics which capture arbitrary application-specific state, on a well-known multicast address. For built-in metrics, *gmond* currently collects and publishes between 28 and 37 different metrics depending on the operating system and CPU architecture it is running on. Some of the base metrics include the number of CPUs, CPU clock speed, %CPU (user, nice, system, idle), load (1, 5, and 15 min averages), memory (free, shared, buffered, cached, total), processes (running, total), swap (free, total), system boot time, system clock, operating system (name, version, architecture), and MTU. User-defined metrics, on the other hand, may represent arbitrary state. *Gmond* distinguishes between built-in metrics and user-defined metrics based on a field in the multicast packets being sent.

Federation in Ganglia is achieved using a tree of point-to-point connections amongst representative cluster nodes to aggregate the state of multiple clusters. At each node in the tree, a Ganglia Meta Daemon (*gmetad*) periodically polls a collection of child data sources, parses the collected XML, saves all numeric, volatile metrics to round-robin databases and exports the aggregated XML over a TCP sockets to clients. Data sources may be either *gmond* daemons, representing specific clusters, or other *gmetad* daemons, representing sets of clusters. Data

sources use source IP addresses for access control and can be specified using multiple IP addresses for failover. The latter capability is natural for aggregating data from clusters since each *gmond* daemon contains the entire state of its cluster.

Data collection in *gmetad* is done by periodically polling a collection of child data sources which are specified in a configuration file. Each data source is identified using a unique tag and has multiple IP address/TCP port pairs associated with it, each of which is equally capable of providing data for the given data source. To collect data from each child data source, Ganglia dedicates a unique data collection thread. Using a unique thread per data source results in a clean implementation. For a small to moderate number of child nodes, the overheads of having a thread per data source are usually not significant.

Ganglia uses the RRD (Round Robin Database) tool to store and visualize historical monitoring information for grid, cluster, host, and metric trends over different time granularities ranging from minutes to years. RRDtool is a popular system for storing and graphing time series data. It uses compact, constant size databases specifically designed for storing and summarizing time series data.

5.3 Scalable storage solutions for the Cloud

5.3.1 Infinispan

Infinispan is an open source data grid platform. It exposes a JSR-107 compatible Cache interface (which in turn extends java.util.Map) that allows to store data in the format of pairs of <keys,values>. While Infinispan can be run in local mode, its real value is in distributed mode where caches cluster together and expose a large memory heap. Distributed mode is more powerful than simple replication since each data entry is spread out only to a fixed number of replicas thus providing resilience to server failures as well as scalability since the work done to store each entry is constant in relation to a cluster size.

Infinispan offers the following functionalities:

- Massive heap and high availability – A system that uses full replication will need to have a copy of the data item on every nodes and will be able to store only the amount of data equals to the memory available in the smallest node. As an example, if the system has 100 blade servers, and each node has 2GB of space to dedicate to a replicated cache, it can only store a total of 2 GB of data. On the other hand, with a distributed grid – assuming that the system was configured to have only one copy per data item – it will be able to store 100 GB of data. If a server fails, the grid creates new copies of the lost data, and puts them

on other servers.

- Scalability – Since data is evenly distributed there is essentially no major limit to the size of the grid, except group communication on the network - which is only used for discovery of new nodes. All data access patterns use peer-to-peer communication where nodes directly speak to each other, which scales very well. Infinispan does not require entire infrastructure shutdown to allow scaling up or down. The user can simply add/remove machines to the cluster without incurring any down-time.
- Data distribution – Infinispan uses consistent hash algorithm to determine where keys should be located in the cluster. Consistent hashing allows for cheap, fast and above all, deterministic location of keys with no need for further metadata or network traffic. The goal of data distribution is to maintain enough copies of state in the cluster so it can be durable and fault tolerant, but not too many copies to prevent Infinispan from being scalable.
- Persistence – Infinispan exposes a CacheStore interface, and several high-performance implementations - including JDBC CacheStores, file system-based CacheStores, Amazon S3 CacheStores, etc. CacheStores can be used for “warm starts”, or simply to ensure that data in the grid survives complete grid restarts, or even to overflow to disk if the system runs out of memory.
- Language bindings – (PHP, Python, Ruby, C, etc.) - The roadmap has a plan for a language-independent server module. This will support both the popular memcached protocol as well as an optimised Infinispan-specific protocol called Hot Rod. This means that Infinispan is not just useful to Java, but also to other applications.
- Management – Running a grid of several hundred servers requires a big management overhead. One way to manage multiple Infinispan instances spread across different servers is to use JOPR which is JBoss' enterprise management solution. JOPR's agent and auto discovery capabilities simplifies the task of monitoring both Cache Manager and Cache.
- Support for Compute Grids – Also on the roadmap is the ability to execute tasks in a remote node of the grid. An application will be able to push complex processing operations towards the server where data is local, and pull back the results. This map/reduce style paradigm is common in applications where a large amount of data is needed to compute relatively small results.

5.3.2 Amazon SimpleDB

SimpleDB is a highly-available key-value storage offering from Amazon. It exposes a simple web

services interface to create, store and query multiple data sets and relies on Amazon's Dynamo in-house storage system.

SimpleDB is able to provide such high levels of availability by using optimistic replication and thus sacrificing consistency under certain failure scenarios. It offers object versioning and application-assisted conflict resolution as ways to address this issue, when it arises.

SimpleDB offers the following functionalities:

- Scalability – In order to scale incrementally, SimpleDB dynamically partitions data over the set of available storage nodes using a circular key space (or “ring”). Partitioning relies on consistent hashing, and adding a physical node simply requires assigning it a random value that represents its position on the ring. Each node has knowledge of all nodes, with the node availability changes being dealt by a gossip-based protocol and regular reconciliation of membership history between random nodes.
- High availability – to achieve high availability and durability, SimpleDB replicates data on multiple, geographically distributed, hosts. When a data item is assigned to a host, it replicates it in the next “healthy” N-1 distinct physical nodes in the ring. This way, each node is responsible for a region of the ring between its own key and the key of its Nth predecessor.
- Data distribution – Each data item is assigned to the node whose key is immediately greater than the data item's hashed key. Since randomness may lead to non-uniform data distribution, each node may be assigned to multiple points in the ring (effectively creating a “virtual-node”). This virtual-node approach allows the system to take into account the heterogenous nature of the underlying infrastructure and guarantees a fairer distribution of the load handled by a node when adding or removing physical nodes (due to failures or routine maintenance).
- Consistency – In order to guarantee replica consistency, the system uses a consistency protocol similar to those used in quorum systems, configurable for both reads and writes. In the event of network partitions or server outages, SimpleDB uses vector clocks to detect conflicts between writes to different replicas, relying on the client to perform semantic reconciliation. Two read consistency options are supported: *eventually consistent reads* are the default and maximize read performance , but may not return the most up-to-date value; *consistent reads* return a result that reflects all writes that received a successful response prior to the read.
- Language bindings – SimpleDB exposes its services through a web services interface. It

also provides SDKs for Java, .Net and PHP.

- Management – Amazon provides an AWS Eclipse plugin that includes full management capabilities for SimpleDB, along with a sample “Scratchpad” application that can be used to manage domains and items within.

5.3.3 Google AppEngine datastore

Google AppEngine's datastore takes advantage of BigTable to provide Google's cloud platform with storage capabilities. Google's BigTable is “a sparse, distributed multi-dimensional sorted map”. The map is indexed by row/column keys and a timestamp, storing raw data as strings. A table's data is stored in lexicographic order of its row key. It relies on the distributed Google File System (GFS) [59] for durability and fault tolerance and the Chubby's file locking services [60] for data consistency,

The row range for a table is dynamically partitioned on full rows, and is called a *tablet*, which acts both as the unit of distribution and load balancing. Every access on a single row key is an atomic operation (regardless of columns touched). BigTable stores multiple versions of data by indexing them with a timestamp. When reading a cell value, the most recent value will be returned, as different versions of the cell will be stored in decreasing timestamp order.

Some of the highlights of AppEngine's datastore are:

- Scalability – Each tablet is assigned to a tablet server by a master server when server addition/expiration is detected. Master servers also deal with tablet load balancing and garbage collection in GFS. Master servers use locking services, file system monitoring and polling on tablet servers to detect available tablet servers. Client data never travels through the master, as it is not used for tablet location. As tables grow, they are automatically split into multiple tablets (each approximately with a 100~200 MB size). When tables shrink, two tablets can also be merged forming a larger tablet.
- High availability – The METADATA table contains the location of all user tablets. All data access patterns will have to access a root tablet that contains the location of all METADATA tablets. The root tablet is never split, guaranteeing that the tablet location hierarchy has always three-levels. Clients cache tablet locations, only accessing the filesystem when a miss is detected. The underlying file system guarantees durability and fault tolerance by keeping a minimum set of replicas for every file-chunk it serves.
- Locality – Since tables are sorted maps, data locality can be boosted by carefully choosing keys so that related data has keys on the same row range. Columns are grouped in column

families, which are the finest grain of access control. Data in a column family is compressed together, and thus all data stored in a column family is usually of the same type. Transaction semantics is possible only for co-located rows, and the system must be advised beforehand on what entities are part of it (by explicitly defining an associated *entity group* when data is first created)

- Persistence – AppEngine stores both log and data files on the distributed Google File System. It uses the SSTable file format, that provides a persistent, immutable ordered map, where both keys and values are arbitrary strings (a tablet is a sequence of SSTables). Commit logs are used to record valid mutations not yet in an SSTable. When a tablet's updates reach a memory threshold, they are converted to a new SSTable and appended to the sequence. Regular compaction of SSTables guarantees that resources aren't wasted and deleted data is purged of the system in a timely fashion.
- Language Bindings – Google provides AppEngine SDKs for Java and Python. These development environments allow for local deployment for testing purposes, although the development and production environments aren't 100% equivalent. The Java SDK allows access to the datastore through standard relational APIs (JDO and JPA) with some limitations inherent to the non-relational nature of the datastore.
- Support for Compute Grids – The datastore can be used with Google's MapReduce, both as input and output.
- Management – The AppEngine Administration Console allows browsing the datastore and managing indexes. It is not possible to directly control the infrastructure only setting upper limits to billing.

5.3.4 GigaSpaces XAP In-Memory Data Grid

GigaSpaces provides an in-memory cache that allows for fast data access, and a distributed cache to deal with high performance and scalability. It relies on a *tuple-space* approach, where a *space* is a distributed shared memory unit, which can be used for data storage as well as messaging. A space provides a simple API that allows users to read, write and remove data from the space as well as to get notified of updates to data. GigaSpaces also exposes a JSR-107 compatible Cache interface (which in turn extends java.util.Map) and a JDBC interface (with limitations due to the mismatch between the space model and the relational model).

The GigaSpaces IMDG provides the following functionalities:

- Massive heap and high availability – Multiple space clustering topologies are available,

from a fully replicated space to a partitioned state with or without backups. It behaves in a similar fashion to Infinispan.

- Data distribution – GigaSpaces uses a routing algorithm based on N routing fields in the data object. If two objects have the same routing field value they will end up in the same partition. With this approach GigaSpaces can determine where to route a data object just by examining the data object. Combining data distribution with replication, the system can provide guarantees on durability and fault tolerance, without compromising scalability (synchronous and asynchronous replication policies are available, with the obvious associated tradeoff in terms of latency/failure resiliency).
- Persistence – In order to ease migration from Hibernate applications, GigaSpaces exposes GigaSpaces and GigaMap interfaces, that can easily map to the equivalent Hibernate API. It is possible to define an external data source, similar to Infinispan's CacheStore, that can be used for “warm starts”, or simply to ensure data in the grid survives complete grid restarts, or even to overflow to disk if you really do run out of memory.
- Language bindings – The core space API is supported in Java, .Net and C++, allowing for interoperability between languages (for instance, write an object in C++ and read it in Java).
- Management – GigaSpaces offers command-line tools, GUI based management console and JMX-based manageable resources.
- Support for Compute Grids – GigaSpaces Data Grid can be upgraded to an Application Server version that fully supports compute grids, where processing units are collocated with the memory unit they use, for a seamless map/reduce style application development.

5.3.5 Cassandra

Cassandra is an open-source highly scalable distributed database that brings together Dynamo's [61] distributed design and BigTable's data model.

It has a proven track record, having been used as the storage infrastructure for Twitter, Facebook and other social websites with large, active data sets.

Most considerations regarding SimpleDB infrastructure apply, with flexibility being added through multiple partitioning strategies (order-preserving hash and also new client-provided ones), topology strategies (how nodes map to physical nodes) and multiple replica placement strategies (taking into account data-center and rack information for each node).

Contrary to SimpleDB, timestamps are provided by clients and are used instead of clock vectors to detect inconsistencies (last timestamp always wins). Every time a read occurs, a check is made to

guarantee consistent replicas. Since the check is only based on timestamps/checksums, the overhead is minimum and in the case of a mismatch allows for the correction of the out of sync replicas: eventually consistent reads perform the read repair after returning a value, whereas consistent reads perform the read repair before returning the value.

From BigTable, Cassandra borrows it's sparse “columnar” data model enhanced with an optional second-level Super Column Families, SSTable disk storage (sequential writes that don't require a previous read) and Hadoop integration (for map/reduce-style processing).

6 Conclusions

This document describes some of the key tools and technologies that the Cloud-TM partners have identified as potential candidates to be integrated into, or influence the development of, the Cloud-TM platform.

The document has surveyed some of the most technologies in the following areas:

- Software Transactional Memories, operating both in a conventional, non-distributed, multicore settings, and in a distributed, shared-nothing, scenario. These technologies represent an important starting point to fulfill the goal of the Cloud-TM project to develop an elastic, self-optimizing, transactional data platform.
- J2EE middleware. The Cloud-TM platform will be implemented in Java and, in order to maximize the chances of widest adoption, it will seek tight integration with existing technologies and standards for the development of large, complex enterprise oriented Java applications. Java 2 Enterprise Edition (J2EE), represents, without dobut, the reference framework in this area.
- Cloud related technologies. This deliverable surveys some of the most popular Cloud technologies in two main areas: i) tools for managing and monitoring of Cloud infrastructures, which are expected to serve as important building blocks for the development of the Autonomic Manager component of the Cloud-TM platform; ii) platforms for efficient data persistence in large scale Clouds, which will influence the development of the distributed storage system of the Cloud-TM middleware.

7 References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2003.

[3] João Cachopo and António Rito-Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In *Prof. of the International Conference on Web Engineering (ICWE)*, pages 297–304, 2006.

[4] N. Carvalho, J. Cachopo, L. Rodrigues, and A. Rito Silva. Versioned Transactional Shared Memory for the FenixEDU Web Application. In *Proc. of the Workshop on Dependable Distributed Data Management (WDDDM)*. ACM, 2008.

[5] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, 2006.

[6] Nuno Carvalho and Jose Pereira. Group communication service for java. http://jgcs.sf.net, 2008.

[7] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Standard – ODMG 3.0*. Morgan Kaufmann Publishers, Inc., Los Altos, USA, 2000.

[8] M. Couceiro, P. Romano, N. Carvalho, and L. RodriguCes. D2STM: Dependable Distributed Software Transactional Memory. In *Proc. of the 15th Pacific Rim Int. Symposium on Dependable Computing (PRDC 09)*, Shanghai, China, November 2009.

[9] M. Couceiro, P. Romano, and L. Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *Proc. of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE Computer Society, 2010.

[10] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[11] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[12] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten. *Weka: A machine learning workbench for data mining.*, pages 1305–1314. Springer, Berlin, 2005.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the Conference on the Management of Data (SIGMOD)*, pages 173–182. ACM, 1996.

[14] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[15] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quema. High throughput total order broadcast for cluster environments. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 549–557. IEEE Computer Society, 2006.

[16] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[17] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[18] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[19] M. A. Hiltunen and R. D. Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany, 2000.

[20] M.F. Kaashoek and A.S. Tanenbaum. An evaluation of the Amoeba group communication system. In *Proce. of the International Conference on Distributed Computing Systems (ICDCS)*, page 436. IEEE Computer Society, 1996.

[21] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, pages 707–710. IEEE, 2001.

[22] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, April 2001. IEEE.

[23] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[24] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. In *Journal of Distributed and Parallel Databases and Technology*, 2003.

[25] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[26] J. Ross Quinlan. Cubist. http://www.rulequest.com/cubist-info.html.

[27] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[28] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *Proc. of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.

[29] Liliana Rosa, Luís Rodrigues, and Antónia Lopes. Building adaptive systems with service composition frameworks. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 754–771. Springer Berlin / Heidelberg, 2007.

[30] David E. Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: the basic theory. pages 1–34, 1995.

[31] Fred B. Schneider. *Replication management using the state-machine approach*, chapter 7, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993.

[32] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, August 1995.

[33] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. Improvements to the SMO algorithm for SVM regression. *IEEE-NN*, 11(5), September 2000.

[34] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. of the 33rd international conference on Very Large Data Bases (VLDB)*, pages 1150–1160. VLDB Endowment, 2007.

[35] Transaction Processing Performance Council. *TPC Benchmark W, Standard Specification, Version 1.8*. Transaction Processing Perfomance Council, 2002.

[36] Transaction Processing Performance Council. *TPC Benchmark TPC-APP, Standard Specification, Version 1.0*. Transaction Processing Perfomance Council, 2004.

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

[38] Dan Grossman , Jeremy Manson , William Pugh, What do high-level memory models mean for transactions?, *Proceedings of the 2006 workshop on Memory system performance and correctness*, October 22-22, 2006, San Jose, California

[39] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II", *Proceedings of the 20th International*

Symposium on Distributed Computing, 2006, pp. 194–208.

[40] Dice, D., Shavit, N.: What really makes transactions fast? *Proceedings of the TRANSACT06 ACM Workshop,* (2006)

[41] Rachid Guerraoui and Michal Kapalka, On the Correctness of Transactional Memory, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP'08)

[42] João Cachopo and António Rito-Silva, Versioned boxes as the basis for memory transactions, *Science Computer Programming*, Volume 36, number 2, 2006.

[43] K. Manassiev, M. Mihailescu, and C. Amza, Exploiting distributed version concurrency in a transactional memory cluster. *In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2006.

[44] Bocchino, Robert L.; Adve, Vikram S. and Chamberlain, Bradford L., Software transactional memory for large scale clusters, *In Proc. of the Symposium on Principles and Practice of Parallel Programming* (PPOPP), 2008.

[45] Kotselidis, C., Ansari, M., Jarvis, K., Lujan, M. Kirkham, C. and Watson, I., DiSTM: A Software Transactional Memory Framework for Clusters, In Proc. of the International Conference on Parallel Processing (ICPP), 2008.

[46] Fenix Framework. https://fenix-ashes.ist.utl.pt/, 2010

[47] Java Transaction API. http://www.oracle.com/technetwork/java/javaee/tech/jta-138684.html, 2010

[48] Data Services Platform. https://www.jboss.com/products/platforms/dataservices/, 2010

[49] Hibernate Core. http://www.hibernate.org/, 2010

[50] Java Message Service. http://www.oracle.com/technetwork/java/index-jsp-142945.html, 2010

[51] Eucalyptus. http://www.eucalyptus.com/, 2010

[52] Open Nebula. http://www.opennebula.org/, 2010

[53] Box Grinder. http://jboss.org/boxgrinder.html, 2010

[54] Lattice Monitoring Framework. http://clayfour.ee.ucl.ac.uk/lattice/, 2010

[55] Ganglia Monitoring System. http://ganglia.sourceforge.net/, 2010

[56] Infinispan. http://www.jboss.org/infinispan, 2010

[57] G. Chockler, I. Keidar, R. Vitenberg, Group communication specifications: a comprehensive study, ACM Computer Survey, Vol. 33, No. 4, 2001

[58] Reliable Distributed Computing with the Isis Toolkit. K. Birman and R. van Renesse, eds. (book) IEEE Computer Society Press, 1994, Los Alamitos, Ca.

[59] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03). ACM

[60] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 335-350.

[61] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07). ACM, New York, NY, USA, 205-220.

[62] Apache Lucene, http://lucene.apache.org, 2010