Towards a simple programming model in Cloud Computing platforms

Jorge Martins, João Pereira, Sérgio M. Fernandes, João Cachopo
IST / INESC-ID
{jorge.martins,joao.d.pereira,sergio.fernandes,joao.cachopo}@ist.utl.pt

***Abstract*—Cloud Computing offers application developers an abstract view of computational resources that can be provisioned on demand over a computer network. This model allows organizations to concentrate on the applications needed to support their core business, instead of having to manage the infrastructure required to run those applications.**

Yet, even though this approach promises very compelling benefits, there is still a lack of programming models capable of bringing the power of parallel programming into the hands of ordinary programmers.

In this paper we tackle this problem by proposing the use of the Fénix Framework as the means of exposing a simple and intuitive programming model for the cloud, while still attaining scalability on par with other Cloud Computing platforms. We claim that may be achieved by integrating the Fénix Framework into the Cloud-TM platform, which is being developed as a self-optimized middleware platform aimed at simplifying the development and administration of applications deployed on Cloud Computing infrastructures.

We validate our claims by describing an initial prototype that provides a simple integration of the Fénix Framework with the persistence tier of Cloud-TM in such a way that it allowed us to run a previously developed benchmark application without changing any of its code to adapt it to the cloud platform.

***Index Terms*—Software Transactional Memory, Persistence, Transactions, Object-Oriented Programming, Rich-Domain Applications, Fénix Framework, Cloud Computing**

I. Introduction

Over the last decade an increasing number of enterprise applications have been developed to support the organizations' processes. These applications typically follow a 3-tiered architecture where the clients make requests to an application server, which, in turn, interacts with a database server. Clients provide the user interface. The application server is responsible for executing business logic and interfacing with the clients. The database server is responsible for the data persistence and ensuring transactional access to it.

This investment on software is usually associated with the corresponding investment in hardware infrastructure to run it efficiently. Simply maintaining and managing all the required hardware and software stack is costly. Furthermore, many of these applications have usage peaks that must be taken into consideration when provisioning for them, but this leads to servers being idle most of the time.

This situation led to the idea of providing hardware infrastructure at a fraction of the cost of owning (and managing) it, and with the possibility to scale up or down the resources allocated based on demand. This is the main idea behind Cloud Computing, and clients may rip huge benefits from this approach. There are several definitions of Cloud Computing. The most consensual is the one defined by *National Institute of Standards and Technology* (NIST) [1]:

> A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Cloud Computing seems to be poised to represent a major paradigm shift in how applications are developed and made available to users. This model allows organizations to concentrate on the applications needed to support their core business, instead of having to manage the infrastructure required to run the applications. Although this approach brings evident benefits to its adopters, most cloud platform vendors provide their own specific set of services, usually with a proprietary API and set of development tools designed to take the most advantage of the cloud platform's qualities.

Cloud platform providers make design decisions taking into account the level of service they aim to provide, the platform's stability, and application scalability and availability. These decisions have direct consequences on the programming model that these platforms provide to developers, often imposing additional restrictions and diverging from the typical enterprise application programming model. Even a proficient enterprise application programmer, when developing cloud-based applications, will need to adjust to the different programming model provided by a given cloud platform. Moreover, once the application is developed for a specific cloud platform, deploying it in another cloud platform typically involves rewriting most of the application, and most likely adjusting to a new programming model.

Our proposal is to provide cloud-based application developers with a programming model/framework closer to the typical enterprise application programming model. This approach builds on previous work on the development of the Fénix Framework [2] and is one of the enabling technologies of the Cloud-TM [3] middleware platform, which aims to simplify the development of services in Cloud Computing platforms, while minimizing administration costs and maximizing scalability and efficiency.

The remainder of this paper is organized as follows. In Section II, we describe some of the main cloud platforms and the challenges they pose to the typical enterprise application developer. Then, in Section IV, we present the Cloud-TM middleware platform. Next, in Section III, we introduce the Fénix Framework and its benefits for developers. In Section V, we provide insight on extending the Fénix Framework towards a cloud platform, while maintaining its programming model. Finally, we conclude in Section VI.

II. Related Work

Given the multitude of cloud platform providers, in this section we shall examine only some of the main players in the field: Google AppEngine, Amazon Web Services, and Microsoft Azure.

Concerns such as application scalability have driven the design of cloud platforms and their storage solutions, both directly influencing how a programmer develops applications.

Google App Engine (GAE) is Google's offering in the *Platform-as-a-Service* (PaaS) realm. It is a software platform for developing and hosting web applications in Google's infrastructure[1], which provides dynamic web serving, along with automatic scaling and load-balancing.

To achieve high scalability, GAE can run multiple replicas of the same application on different *Application servers*. This way it is simple to scale up or down an application automatically, based on its usage, simply by adjusting the number of Application servers running it. This approach is possible because all requests are load-balanced and forwarded to *Frontends* that are responsible for routing the requests to available Application servers.

Persistence is attained through the use of the *datastore*. The datastore is a schemaless object database, including its own query engine and support for transactions. Each data object in the datastore is known as an *entity*. Objects in the datastore are organized into *entity groups*. Datastore transactions can manipulate objects only of the same entity group, providing Serializable isolation level inside transactions. GAE's datastore ability to scale comes from the possibility of assigning different entity groups to different network nodes.

It is the programmer's responsibility to decide which objects go into which entity group. This decision becomes quickly overwhelming in the context of an enterprise application with a rich domain model, and large complex object graphs. The naïve solution is to place all objects in the same entity group, but it does not scale: all datastore accesses would be performed on the same node. Any other approach requires the programmer to decouple, in a sensible way, what is a tightly coupled object graph.

Recently, Google introduced the possibility to perform cross-group transactions, although limited to five entity groups and incurring in performance penalties. This possibility could help reduce the

[1]http://code.google.com/appengine

effect of any decision regarding breaking the object graph into entity groups.

Amazon pioneered the cloud offerings and has since provided a multitude of services that add up to the *Amazon Web Services* (AWS).[2]

Amazon Elastic Beanstalk takes advantage of Amazon's *Elastic Compute Cloud* (EC2) coupled with Elastic Load Balancing and Auto Scaling services to provide a scalable Java-based cloud platform. The programmer can specify how an application should scale given usage patterns, schedules, or other criteria. As with Google's approach, multiple replicas of the same application will run on different EC2 instances, with traffic being redirected to different ones by the load balancer.

A scalable, high-availability database is provided by the SimpleDB service. SimpleDB is a schemaless object database, including its own query engine and automatic indexing facilities. Each data object in SimpleDB is known as an *item* and is replicated automatically in multiple different geographic locations. Items in SimpleDB are organized into *domains*. To be able to scale up its application, the developer also needs to partition the application data and requests into multiple domains. Transactional support is limited to accesses to a single item. For enterprise applications with rich domain models, where a single business operation may read, create or modify multiple objects, SimpleDB does not provide any guarantees regarding the whole operation.

Microsoft has developed Windows Azure Platform[3] as its own Cloud offering. With Azure, Microsoft provides a set of developer services that allow building and hosting applications on Microsoft's infrastructure.

Applications are built from one or more roles, a formalization of the logical partition of an application. Windows Azure provides three types of roles: *web roles*, which are intended for logic that interacts with the outside world via HTTP and IIS; *worker roles*, whose interactions with the outside world are not limited to HTTP and may be used to perform complex tasks; and *VM roles*, which allow organizations to pre-configure Windows 2008 R2 server images and deploy them to run on the cloud.

[2]http://aws.amazon.com

[3]http://www.microsoft.com/windowsazure

The Windows Azure programming model stipulates that an application is built from one or more roles, that it runs more than one instance of each role, and that it behaves correctly in case of an instance failure. These rules guarantee that administration tasks and operational hazards are not disruptive to the application (OS patches, application upgrades or hardware failures will likely affect the application performance, but not its availability) and that applications benefit from on-demand scalability (through a web portal, configuration, or specific API).

Windows Azure provides storage services designed to handle very large data sets: *tables* to store structured data sets using a non-relational approach, and *blobs* for binary data. Every table/blob is automatically replicated three times, in distinct physical machines.

The Table storage is a schemaless object database, including its own query engine and support for transactions. Each data object in the table storage is known as an *entity*. Entities are organized into *tables* that may be partitioned across multiple storage nodes. Transactions may manipulate entities only within the same table and the same partition. Additionally, an entity may appear only once within a transaction, and only one operation may be performed against it. Furthermore, read operations must not be mixed with write operations. Partitioning is controlled by the programmer through a PartitionKey assigned to each object. Similarly to GAE's datastore, this policy places the responsibility of correctly splitting a large and complex domain object graph into partitions for scalability. Furthermore, the limitation on read/write operation mixing within a transaction imposes additional restrictions on how complex business logic can be implemented.

III. FÉNIX FRAMEWORK

The Fénix Framework[4] [2] has matured in the context of the FénixEDU project with the goal of simplifying the development of Java enterprise applications. The core idea is to shift responsibility for transactional control from the persistence tier to the application server tier. Persistence is still provided by the persistence tier.

[4]http://fenix-ashes.ist.utl.pt/trac/fenix-framework

The Fénix Framework uses the JVSTM [4] to provide transactional control and integrates it with a relational database system. The JVSTM is a multi-versioned *Software Transactional Memory* (STM) [5] that provides in-memory transactional support right in the application server tier. JVSTM guarantees strict serializability for all committed transactions, while providing efficient reads and ensuring that read operations never conflict.

Transaction support requires transaction-aware domain objects. To this purpose, JVSTM provides a domain-specific language [6] that can be used to describe the structural aspects of an application's domain model—the *Domain Modeling Language* (DML) [4], [7]. The DML uses a Java-like syntax to describe entities in terms of their attributes, and relationships between entities. A developer using the Fénix Framework would use the DML to describe the application's domain model and use the DML compiler to generate the source code that implements the structural aspects of domain classes in a transaction-safe way; the developer would then extend these classes with behavior using plain Java.

To load persistent objects, the Fénix Framework queries the database, performing transparent mapping from the underlying relational model to the domain model specified through the DML. When persisting objects, the Fénix Framework extends the JVSTM *commit* operation to persist changes to the database, again performing transparent mapping from the domain model to the relational model.

The Fénix Framework provides a domain object cache to reduce the performance penalty of accessing the database. An object request is always checked first against the cache, and only if not found there will it be loaded from the database (and stored in the cache for future lookups). This cache stores domain objects for as long as possible, and given that domain objects are transaction-safe, they exhibit the useful property of having their own identity: There will be at most one instance in memory of any persistent object, regardless of the number of concurrently running transactions.

IV. CloudTM

The Cloud-TM platform is a middleware that is being developed for service implementation in Cloud Computing infrastructures, and it includes the following goals [8]:

- To offer a simple and intuitive programming model for the implementation of services in Cloud Computing platforms. In particular, by relieving developers of the burden of dealing with low-level mechanisms such as distribution, persistence, and fault-tolerance; instead allowing them to concentrate on delivering added business value.
- To minimize operational costs by automating provisioning of resources from the underlying Cloud platforms while guaranteeing a user-specified Quality-of-Service.
- To maximize scalability and efficiency of provided services through self-tuning capabilities that can deal with fluctuations both in allocated resources and workload.

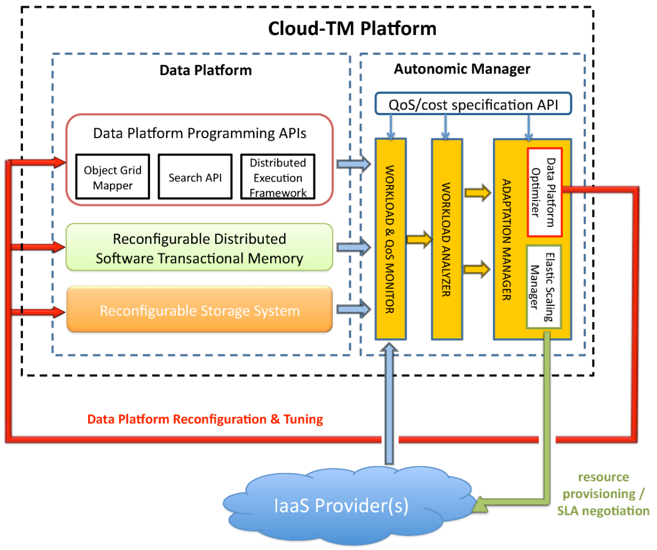
Cloud-TM Platform
Data Platform
Autonomic Manager
QoS/cost specification API
Data Platform Programming APIs
Object Grid Mapper
Search API
Distributed Execution Framework
Reconfigurable Distributed Software Transactional Memory
Reconfigurable Storage System
WORKLOAD & QoS MONITOR
WORKLOAD ANALYZER
ADAPTATION MANAGER
Data Platform Optimizer
Elastic Scaling Manager
Data Platform Reconfiguration & Tuning
IaaS Provider(s)
resource provisioning / SLA negotiation

Figure 1. Cloud-TM architecture

Figure 1 provides an overview of the proposed architecture for Cloud-TM[9]. It encompasses two main blocks: a Data Platform and an Autonomic Manager.

The Data Platform is responsible for retrieving, manipulating, and storing data across a dynamic set of distributed nodes that are elastically acquired from one or more underlying *Infrastructure-as-a-Service* (IaaS) Cloud providers. It will expose a set of APIs, denoted as *Data Platform Programming APIs* in Figure 1, aimed at increasing the productivity of Cloud programmers in two ways:

- By allowing ordinary programmers to read/write data from/to the Data Platform using the familiar abstractions provided by the object-oriented paradigm, such as inheritance, polymorphism, associations.
- By allowing ordinary programmers to take full advantage of the processing power of the Cloud-TM platform via a set of abstractions that will hide the complexity associated with parallel/distributed programming, such as load balancing, thread synchronization, scheduling, or fault-tolerance.

The main component of the Data Platform, is a highly scalable, elastic and dynamically Reconfigurable Distributed Software Transactional Memory (RDSTM). Red Hat's Infinispan is being used as a starting point for developing this essential component of the Cloud-TM platform.

Infinispan is a recent in-memory transactional data grid designed from the ground up to be extremely scalable. Infinispan is being extended with new algorithms both for data replication and distribution, and real-time self-tuning schemes aimed at guaranteeing optimal performance even in highly dynamic Cloud environments.

The lowest level of the Data Platform provides abstractions that allow state to be persisted over a wide range of heterogeneous durable storage systems, from local/distributed filesystems to Cloud storages.

The Autonomic Manager is the component in charge of automating the elastic scaling of the Data Platform, as well as of orchestrating the self-optimizing strategies that will dynamically reconfigure the data distribution and replication mechanisms to maximize efficiency in scenarios entailing dynamic workload.

Its topmost layer will expose an API allowing the specification and negotiation of QoS requirements and budget constraints. The Autonomic Manager will collect information not only about heterogeneous system-level resources (such as CPU, memory, network and disk), but will also characterize the workload of each of the components of the Data Platform and their efficiency.

This information is then processed by the Workload Analyzer, which will then inform the Adaptation Manger. Finally, the Adaptation Manager is responsible for self-tuning the various components of the Data Platform and control the dynamic autoscaling mechanism with the ultimate goal of meeting QoS/cost constraints.

The above description of the qualities desired for the Data Platform Programming APIs makes the Fénix Framework the perfect candidate to implement it. It provides programmers with a language that has a familiar syntax to describe a rich domain model; and a minimalistic API that provides the single new notion of atomic operation (or transaction), making it easier to reason about parallel programs.

Integration of the Fénix Framework with the remainder of the Cloud-TM platform needs to occur at two distinct levels: changing the Fénix Framework implementation to rely on Infinispan as the persistence tier; additionally building a new DML compiler that generates code that takes into account the specificities of the platform, such as providing monitoring information to the Workload Monitor or providing different data layouts to adjust to different memory and concurrency requirements.

This way, developing an application using the Cloud-TM platform would be similar to develop a standalone enterprise application using the Fénix Framework, but would take advantage of the platform's elasticity and self-tuning characteristics with little to no extra effort. Contrary to any of the platforms described in Section II, the Fénix Framework imposes no restrictions on what domain objects can participate in a transaction, as long as they have been defined in DML, and it provides stronger isolation level to business operations (strict serializability). In addition, the self-tuning characteristics of Cloud-TM can be used to increase often-related data locality, thus increasing scalability in an autonomic way, with no programmer intervention or explicit data partitioning.

V. Validation

The Fénix Framework described in [2] uses a relational database system to store the state of the application persistently. The Fénix Framework provides a strictly serializable semantics to the applications' business transactions. This semantics relies on the STM used by Fénix Framework. The *commit* operation of transactions had to be extended to store persistently the changes made by a transaction that can commit because, usually, STMs do not handle persistence of data. With this extension, once the committing transaction has been validated by the STM, the changes made in the context of this transaction are stored in the relational database system. If by some unexpected reason an error happens in the process of writing the changes to the relational database system then the committing transaction is aborted. Otherwise, we can commit the transaction and make visible its effects in main memory.

To validate our claims, we made a simple implementation of the Fénix Framework where we use Infinispan as our persistence tier instead of a relational database system. To keep this initial implementation simple, we assume that there is a single instance of the application server running. This way, we did not need to implement a mechanism to synchronize the transactions running in different application servers. There has been, however, a previous effort with another Fénix Framework implementation (D2STM [10]) that can work in a distributed environment. Future implementations of the Fénix Framework using Infinispan will integrate that work to allow multiple instances of the application server to detect conflicts between them. The changes made in Fénix Framework to use Infinispan are described as follows.

To reduce the number of accesses to the persistence tier, the Fénix Framework uses a domain object cache, which keeps the domain objects in memory as long as possible. This cache is shared by all threads of the application server. One of the goals of the Fénix Framework was that read-only transactions should access the persistence tier only when the desired version of a domain object is not available in memory, which should not happen often if most of the application's data fit in main memory. Thus, once loaded into main memory, the versions of a domain object should remain in main memory until they are no longer needed by currently executing (or future) transactions or the Java garbage collector needs space that it cannot find in any other way. In the last case, the versions of the domain objects that are not being used in any running transaction may be removed from the cache by the garbage collector[5]. Due to this reason, the Java garbage collector may remove a version of a domain object from the cache that may be necessary for a current transaction. This way, it is not enough to just store the last version of each domain object since each transaction needs to access a consistent view of the world (the whole set of application domain objects) to support the serializability property of transactions and it may have to access a version other than the last version of a domain object. The Fénix Framework described in [2] relies on how the specific underlying relational database implements its transactions to guarantee this property. Our solution to cope with this problem was to store all versions of each domain object in Infinispan.

The advantage of storing all versions of each domain object is that we can run a read-only transaction somewhere in the past and view the state of the system at that point in time. This can be useful for auditing purposes. However, with this approach we have an unbounded growth of the storage space required for storing all versions of each domain object. If this becomes an issue, we can remove the versions of domain objects that are no longer needed from the persistence storage.

The Fénix Framework keeps a set of versions for each accessed domain object in main memory. To limit the amount of memory needed to store all these versions, there is an algorithm that removes all versions that are no longer needed, taking into account the set of running transactions. This algorithm is executed asynchronously whenever a transaction finishes. We can adapt this algorithm to also delete the removed versions from the persistence storage.

Infinispan offers a cache interface to store key/value pairs. In Fénix Framework, each domain object has a unique object identifier (OID) and it may have several versions (recall that Fénix Framework uses a multi-version STM) and we need to have a way of storing an object and all of its versions using the cache interface. To do this, each version of a domain object is stored as an entry in the cache having the pair (OID of the domain object, current version number) as the key and the triple (current version number, previous

[5]In our current implementation, we rely on Java's SoftReferences for this behavior.

version number, serialization of domain object) as the value. Each version of a domain object stored as an entry in a Infinispan cache has the information needed to compute the key of the previous version. This key is equal to the OID of the domain object plus the previous version number stored in the value of the entry. We also have an extra entry in the cache per object that holds the most recent version of the object. The key of this entry is just the OID of the object and its value has the same format as for the other entries. This entry, designated as the *root* entry, gives us the entry point for looking up in Infinispan the correct version of a domain object for a given transaction, i.e., the version with the highest version number that is smaller than the version number associated with the transaction. When it is necessary to get from Infinispan the correct version of a domain object for a given transaction, we just need to traverse the list of entries associated with this domain object until we find an entry whose value has a current version number smaller than the version of the transaction. The first element of this list is the root entry associated with the domain object. Finally, we write the changes of a committed transaction in Infinispan in the context of an Infinispan transaction. This way, we can assure that the process of storing the changes in the persistence tier is atomic (as desired), i.e., all changes are written in the persistent tier or none is made.

With this proof-of-concept implementation of the the Fénix Framework we were able to run an existing benchmark (more specifically, the TPC-W benchmark [11]) without having to change any of the benchmark's code.

The tests were performed running a single application server in a machine with a NUMA architecture, built from four AMD Opteron 6168 processors. Each processor contains 12 cores, thus totaling 48 cores. The system had 128GB of RAM, more than enough to keep data from all processes (clients, application server and Infinispan) loaded in main memory. We tested using Java 6 with HotSpot(TM) 64-Bit Server VM (19.1-b02). The application server was installed as a web application on Apache Tomcat 6. We populated the database with 1,000 books and 28,800 clients,[6] and tested the application with the number of concurrent clients ranging from 1 to 10.

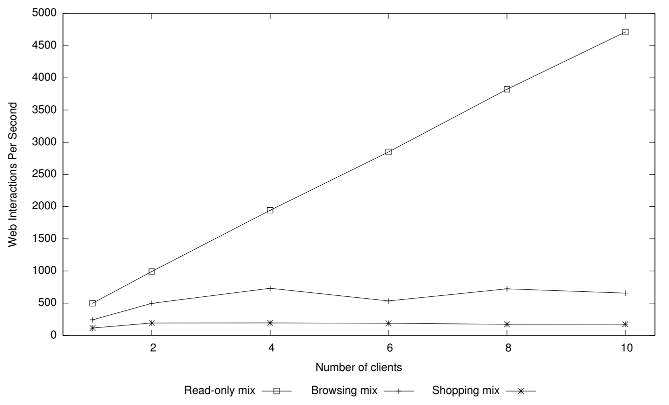

Figure 2. Throughput for a single application server, varying the number of concurrent clients.

Figure 2 shows that the Infinispan-version of the Fénix Framework scales with the number of clients for the *Read-only mix* scenario but that does not happen for the other two scenarios, specially for the *Shopping-mix* scenario where the measured WIPS remains almost constant for a number of clients greater than 2. This is due to the fact that the STM used by the Fénix Framework uses a global lock for committing successful transactions. The cost in time to write to Infinispan the objects modified by a transaction is high. Since this writing only happens during the commit process of a successful transaction this means that the time to execute a transaction is dominated by the time to commit the transaction. Therefore, when we have several concurrent transactions that modify the system most of the time they are waiting that some other transaction finishes its commit. This problem does not happen in the *Read-only mix* scenario since in this case the transactions do not modify the state of the system. There has been ongoing work on the JVSTM to make it lock free [12]. In the future we expect to incorporate this updated version of the JVSTM on our implementation of the Fénix Framework.

[6]According to the TPC-W specification, the database must be populated with $2880 \times c$ clients in order to use up to c concurrent clients.

VI. Conclusion

In this paper we proposed the simple programming model provided by the Fénix Framework as a foundation for a programming model for the new generation of enterprise applications being developed to run in the cloud.

We briefly described the work that we have been doing in the Cloud-TM platform to integrate our previous work on the Fénix Framework into the remainder of the Cloud-TM platform. Specifically, how to change the Fénix Framework so that it targets Infinispan as the underlying persistence layer.

We have succeed in doing this while maintaining the same API of the Fénix Framework, thereby allowing us to run the TPC-W benchmark on top of the Infinispan-version of the Fénix Framework without having to change any of the benchmark's code.

Even though our work is still an initial prototype, it shows already some very promising results that we believe will help address some of the current challenges that the community is facing:

- How to offer a simple and intuitive programming model for the implementation of applications deployed in Cloud Platforms?
- How to minimize operational costs by automating resource provisioning from one or more Cloud providers, while meeting user-specified Quality of Service?
- How to maximize scalability and efficiency of applications through self-optimizing capabilities that are able to deal with workload and resource allocation fluctuations?

We argue that the Fénix Framework already provides the required qualities (in a typical enterprise application environment) to address the first challenge, and that integrating it as Cloud-TM's Object Grid Mapper can bring all of the qualities to Cloud-based applications.

Finally, we expect to rely on work still in progress in the Autonomic Manager of the Cloud-TM middleware to provide the required self-tuning capabilities that will allow our approach to fully deliver on what seem promising results.

Acknowledgment

The work described in this paper would not have been possible without the contribution of many people that have been involved in some way or another with the development of the Fénix Framework and of the FénixEdu system. These include the excellent development and systems' administration teams at CIIST, and the members of the Software Engineering Group at INESC-ID.

Also, this work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, and by the Cloud-TM project, which is co-financed by the European Commission through the contract number 257784.

References

[1] P. Mell and T. Grance, "NIST Definition of Cloud Computing," http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf, Jan. 2011.

[2] S. M. Fernandes and J. Cachopo, "Strict serializability is harmless: A new architecture for enterprise applications," in *SPLASH'11 Companion*, ser. SPLASH '11. New York, NY, USA: ACM, Oct. 2011.

[3] CloudTM Consortium, "Cloud·TM: A novel programming paradigm for the cloud," http://www.cloudtm.eu/, Jun. 2010.

[4] J. Cachopo, "Development of rich domain models with atomic actions," Ph.D. dissertation, Instituto Superior Técnico/Universidade Técnica de Lisboa, Sep. 2007.

[5] J. Cachopo and A. R. Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, pp. 172–185, Dec. 2006.

[6] M. Fowler, *Domain-Specific Languages*, ser. Addison-Wesley Signature Series. Addison-Wesley Professional, 2010.

[7] J. Cachopo and A. R. Silva, "Combining software transactional memory with a domain modeling language to simplify web application development," in *Sixth International Conference on Web Engineering*. Palo Alto, USA: ACM, Jul. 2006, pp. 297–304.

[8] E. Bernard *et al.*, "D1.2: Enabling technologies report," CloudTM Consortium, Tech. Rep., Nov. 2010.

[9] ——, "D2.1: Architecture draft," CloudTM Consortium, Tech. Rep., Jun. 2011.

[10] N. Carvalho, J. Cachopo, L. Rodrigues, and A. Rito Silva, "Versioned transactional shared memory for the FenixEDU web application," in *Workshop on Dependable Distributed Data Management (WDDDM)*. New York, NY, USA: ACM, 2008.

[11] H. W. Cain and R. Rajwar, "An architectural evaluation of java tpc-w," in *In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 229–240.

[12] S. M. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM SIGPLAN, Feb. 2011, pp. 179–188.