A Generic Framework for Replicated Software Transactional Memories

Nuno Carvalho, Paolo Romano, Luis Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa
Lisbon, Portugal
nonius@gsd.inesc-id.pt, romano@inesc-id.pt, ler@ist.utl.pt

Abstract—Software Transactional Memory (STM) has emerged a powerful abstraction for managing access to shared data. Therefore, it is no surprise that a handful of different STM replication schemes have been proposed in the last recent years. In this context, we propose an architecture that facilitates the integration and execution of multiple replication techniques in a single, coherent, middleware infrastructure. This paves the way towards the development of autonomic mechanisms, able to select in runtime the most appropriate replication technique for the workload at hand.

Keywords—Software Transactional Memory; Software Architecture; Fault Tolerance.

I. INTRODUCTION

Fostered by the current architectural trend that has lead to the pervasive adoption of multi-core CPUs, the interest towards Software Transactional Memory (STM) systems has grown considerably over the last years. STMs represent an attractive solution to spare programmers from the pitfalls of conventional lock-based thread synchronization mechanisms. STMs rely on the abstraction of atomic transactions to regulate concurrent accesses to shared memory and free programmers from the complexity of lock-based synchronization schemes, drastically simplifying the development and verification of concurrent programs, enhancing code reliability and productivity [1].

Over the last years, a wide body of literature has been developed in the area of STMs and, recently, real-world STM-based applications have started to be deployed in production systems [2]. Since replication represents a key instrument to enhance the availability, failure resiliency, and scalability, several STM replication schemes have been proposed in the last recent years [3], [4].

The performance of the applications can be improved by using the replication protocol and local STM mechanism that better fits the generated workloads. Therefore, system engineers willing to build applications for replicated STM systems are faced with a very difficult choice: what combination of replication protocol and concurrency control algorithm should be picked in order to maximize the performance of a given application, deployed on a specific hardware platform? The problem is further exacerbated by the fact that workloads can dynamically change over time, for instance, if the population of users grows, or as the system is enriched with new functionalities. Unfortunately, given the high heterogeneity of APIs exposed by existing systems for replicated STMs, the choice of the replicated STM platform is currently a committing one, locking the user application to specific middleware solutions.

In order to tackle these issues, in this paper we present GenRSTM, a Generic framework for Replicated STMs. GenRSTM has been designed in order to support, in an efficient and modular fashion, a wide range of heterogeneous algorithms across the various layers composing the software stack of a replicated STM platform, and specifically (i) replica consistency, (ii) local concurrency control and (iii) group communication system. Flexibility is achieved via a set of neat, reflective interfaces [5], [6], which allow the replication manager to be notified of information/events reflecting the internal state of the local STM (such as the read-set/write-set of committing transactions, or the activation of new local transactions), as well as to alter the state of (possibly heterogeneous) local STMs on the basis of the outcome of the selected replica coordination protocol. Efficiency is achieved via the adoption of the observer [7] software design pattern, which allows to restrict the notifications exchanged between the STM and replication modules exclusively to the ones that are strictly needed by the specific configuration of the entire middleware stack.

The internal software architecture of GenRSTM relies on the Inversion of Control [7] and dependency injection [7] design patterns. By effectively separating the development of functional behavior from dependency resolution, this allows not only to enhance reusability by reducing coupling among software modules. It also allows to reduce the complexity of the individual components by sparing developers of new replication/STM modules from developing boiler-plate code to hard-code dependencies.

Overall, GenRSTM allows system administrators to seek optimal performance as a function of the workload/deployment scenario by reconfiguring the replicated STM middleware platform, in a transparent fashion for the user level application. Thanks to its modular and extensible design, and by making available a number of building blocks required by replicated STM solutions, GenRSTM aims at simplifying the development of new STM replication algorithms and at integrating the results from the growing community of researchers working in this area.

The paper is organized as follows. Section II summarizes...
the related work. Section III describes the proposed architecture, including the minimum set of interfaces to achieve a generic framework. Finally, Section IV concludes this paper.

II. RELATED WORK

DSTM2[8] represents, to the best of our knowledge, the first generic framework proposed to simplify and homogenize the development and comparison of alternative non-distributed STM schemes. A more recent, also non-distributed, generic STM framework is Deuce [9], which, unlike DSTM2, also allows to accommodate implementations of multi-versioned STMs. Unlike DSTM2 and Deuce, GenRSTM targets replicated STMs, distributed across a set of nodes.

The closer existing solution to our proposal is DiSTM[10], which is the only framework for distributed STMs we are aware of. With respect to GenRSTM there are a number of relevant differences. First, being based on the aforementioned DSTM2, unlike our proposal, DiSTM can locally support exclusively single-versioned STMs. Further, the focus of GenRSTM is on replicated STMs, whereas DiSTM provides support for distributed, but not replicated, STMs. Finally, by relying on a Generic Group Communication Service (jGCS), our framework allows to seamlessly integrate with a wide range of different communication paradigms.

III. THE GENRSTM ARCHITECTURE

In this section we present the proposed architecture. The GenRSTM architecture focus on the following goals:

- **Goal #1:** Simplify the development and testing of new replication protocols and STMs: It is a simple way to test new protocols, by using useful building blocks which satisfy common requirements of replication protocols and STMs.

- **Goal #2:** Provide high decoupling between the architecture building blocks: It allows each building block to be implemented independently by using well defined interfaces.

- **Goal #3:** Support multiple implementations of the architecture building blocks: It allows each building block to be composed with other existing components, so that the system can be tuned to achieve the best performance of a specific workload and network characteristics.

In the following paragraphs, we show how these goals are achieved.

A. Overview

The components of a node of the generic platform, depicted in the Figure 1, are structured into three main logical layers. The bottom layer is a Group Communication Service (GCS) [11] which provide two main building blocks: view synchronous membership [12], and a set of interfaces that provide communication services. The ordering and fault tolerance guarantees offered by these services depend on the configuration of the underlying implementation. For instance, full replication protocols based on certification require the underlying toolkit to provide an Atomic Broadcast (AB) service [13]. The architecture uses a generic group communication service for Java (jGCS) [14].

The core component of the generic architecture is represented by the Replication Manager (RM), that implements the distributed coordination protocol required for ensuring replica consistency. The RM interacts, on one side, with the GCS layer and, on the other side, with a local instance of a STM. Finally, the top layer of the architecture is a wrapper API that intercepts the application level calls for transaction demarcation (e.g., to begin and commit transactions) and read/write operations. This approach allows to extend the classic STM programming model, allowing the concurrent execution of an arbitrary number of threads on each replica.

The integration of an STM within this generic architecture requires the implementation of five main extensions to allow the RM to interact with the STM without requiring re-implementing several components strictly related to the STM itself. More specifically, the STM should export its behavior by means of a reflective [5] mechanism including the procedures needed by the replication protocols, namely:

- add listeners for the transaction demarcation procedures (begin, commit and abort) and to read/write operations that can be used by the RM. This allows the RM to trigger the distributed coordination protocol required for ensuring replica consistency;
- extract information concerning internals of the transaction execution, i.e., its read-set, write-set, and snapshot ID;
- explicitly trigger the transaction validation procedure, that aims at detecting any conflict raised during the execution phase of a transaction \( T_x \) with any other (local or remote) transaction that committed after \( T_x \) started;
- atomically apply the write-set of a remotely executed trans-
action and simultaneously increasing the STM’s timestamp; – permit cluster wide unique identification of the memory objects created and updated by (remote) transactions. This is achieved by tagging each STM object with a unique identifier. A variety of different schemes may be used to generate universal unique identifiers (UIDs), as long as it is possible to guarantee the cluster-wide uniqueness of UIDs generated independently at each replica.

Some of these procedures already exist in most STM systems, they just need to be exported so they can be used by any external component. The access to these procedures allows for a more efficient replication solution, since the replication protocols will use the mechanisms natively implemented by the STM. In the case of the presented architecture, they will be used by the RM.

Each component of GenRSTM needs to have well defined interfaces. This allows that each component can be developed independently and composed with several compatible implementations. The interfaces needed in the system are the Distributed STM API, the Actuator API, the Reflective API and the Group Communication Service API. For the latest, GenRSTM uses the already existing Generic Group Communication Service for Java (jGCS) [14]. This service is composed by a set of interfaces to send and receive messages to a group of processes. jGCS can be configured for using several communication paradigms (e.g. Atomic Broadcast or epidemic multicast), depending on the requirements of the replication protocol. The other APIs are described in the next sections.

B. APIs Towards the Programmer

The STM Runtime Context API encapsulates the STM implementation and provides to the application methods to begin a new transaction, commit or abort a transaction, and create new objects. To create new objects, the API exports a factory that is implemented by the specific STM. The transactional objects must be serializable and uniquely identifiable among the distributed system. The API provides a factory to generate unique identifiers.

GenRSTM follows an Object Oriented model and was targeted to provide field granularity in its transactions. This is materialized by having each field encapsulated within a Box. A Box has methods to read the latest value, write a new value, get the data version of the latest committed value and commit the latest written value, which means making it visible to other transactions. Each Box needs also to be uniquely identified. A Box can contain a primitive value or a reference to a transactional object. Note that a Box could be replaced by a bytecode rewriting mechanism, but the practical result is the same, since the developer still needs to mark somehow the fields in the objects that must be replicated.

With this simple, but intuitive, interface the framework provides a generic mechanism to shield applications from the details of the STM being used, either replicated or not. This contributes to achieve the Goal #1.

C. APIs Between RM and STM Layers

An STM creates and executes transactions, reflecting its internal behavior to the lower layer. This is abstracted by the Transaction interface. Implementations of this interface keep all the information needed by the underlying protocols, such as a ReadSet, a WriteSet, a transaction identifier (ID) and a data snapshot timestamp. A read-set is composed by a list of Box IDs. A write-set is composed by a list of the pairs (BoxID, value), where the value can be a primitive value, an ID of an existing distributed object, or a Serializable version of a new Object. If the object is new (was created inside the specified transaction), the whole object must be included in the write-set, so it can be sent to the remote replicas.

The Transaction, ReadSet and WriteSet interfaces are used in the Actuator and Reflective APIs. The Reflective API reflects the behavior of the transaction to the RM. This is achieved by adopting the Observer Design Pattern. The RM can implement the TransactionListener and/or the OperationListener interfaces, and register them on the STM layer. This way, the RM is notified on the TransactionListener when a new transaction is started (onBegin(Transaction t)), a transaction is starting the commit phase (onCommitting(Transaction t)) and the transaction finished, either by committing or aborting (onFinished(Transaction t, boolean committed)). If the RM registers also the OperationListener, it is notified when a Box is read (onRead(Transaction t, Box b)) and is written (onWrite(Transaction t, Box b). These methods pass the execution control to the lower layer (RM) which can pause the execution of the transaction (e.g. if it needs to wait for a message to be delivered).

The Actuator API is used by the RM to act on the STM, for two operations: atomically apply the write-set of a remote transaction and certify a (local or remote) transaction. The apply(WriteSet ws) method is used only to apply a remote and valid transaction and should be implemented by the STM, using its already existing internal mechanisms. It exposes new values on the Boxes and increments the data version, if the STM needs to maintain one. The validate() method is also implemented and exposed by the STM and should be able to validate any transactions’ read-set against the current memory state, returning true if the transaction is valid.

D. Enhancing Decoupling Between Components

To achieve Goals #2 and #3, the framework adopted not just the previously presented interfaces, but also the Inver-
sion of Control (IoC) and Dependency Injection (DI) design patterns. IoC allows to achieve a high degree of decoupling among the previously described architecture components. By adopting the IoC design pattern, this framework allows the accommodation of several implementations of each module of the architecture (STM, RM and GCS). This has several benefits. First of all, the programmer is able to implement each building block independently, focusing only on the task of that block. For instance, when a programmer is implementing a new replication protocol, he/she does not have to deal with ensuring message ordering or the details of applying a write-set of a remote transaction on the STM layer. Secondly, the several components can be reused and composed for a specific scenario or application. Using the same example, if a programmer needs to implement a new replication protocol, this new protocol can be composed with already existing implementations of STMs and Group Communication toolkits. Finally, using IoC means that replacing one component of the architecture by another that ensures the same type of guarantees will have no side effect on other components. The only visible effect of replacing a component is the resulting performance change, when executing the same application with a different setup.

For each building block, an implementation needs to define its dependencies and the system is executed after filling all those dependencies. This is achieved in the presented framework by making use of the DI design pattern. DI is used to ensure that all the dependencies among modules are met. Upon its instantiation, each module is provided a reference for the specified dependencies, avoiding the explicit creation of the required modules. The current prototype uses the Google Guice\(^1\) DI framework. Note also that DI is used only when all the objects are instantiated, affecting the performance of the system only in the bootstrap. The performance of the system is not affected during its execution.

IV. CONCLUSIONS

GenRSTM is a framework that, in its current version, is able to integrate different replication schemes and STMs, being completely transparent to the application. GenRSTM was implemented in Java and is available for download as open source software (http://aristos.gsd.inesc-id.pt). It allows the configuration of the system with an optimal setup for the given workload and network characteristics. This architecture simplifies the development and evaluation of alternative replication protocols, by encapsulating the three key components of a replicated STM that communicate through well defined interfaces. With the presented generic architecture new results can be integrated, serving also as a tool that helps the community to easily prototype and test new replication algorithms.

\(^1\)http://code.google.com/p/google-guice/

ACKNOWLEDGMENTS

This work has been partially supported by the project “Cloud-TM” (co-financed by the European Commission through the contract no. 257784), the FCT project ARISTOS (PTDC/EIA-ElIA/102496/2008) and by FCT (INESC-ID multiannual funding) through the PIDDAC Program Funds.

REFERENCES