Chasing the Optimum in Replicated In-memory Transactional Platforms via Protocol Adaptation (Regular Paper)

This submission has been cleared through the authors' affiliation

***Abstract*—Replication plays an essential role for in-memory distributed transactional platforms, such as NoSQL data grids, given that it represents the primary mean to ensure data durability. Unfortunately, no single replication technique can ensure optimal performance across a wide range of workloads and system configurations. This paper tackles this problem by presenting MORPHR, a framework that allows to automatically adapt the replication protocol of in-memory transactional platforms according to the current operational conditions. MORPHR presents two key innovative aspects. On one hand, it allows to plug in, in a modular fashion, and to regulate the switching between arbitrary replication protocols. On the other hand, MORPHR relies on state of the art machine learning techniques to autonomously determine the optimal replication in face of varying workloads. We integrated MORPHR in a popular open-source in-memory NoSQL data grid, and evaluated it by means of an extensive experimental study. The results highlight that MORPHR is accurate in identifying the optimal replication strategy in presence of complex, realistic workloads, and does so with minimal overhead.**

I. INTRODUCTION

With the advent of grid and cloud computing, in-memory distributed transactional platforms, such as NoSQL data grids [1], [2] and Distributed Transactional Memory systems [3], [4], have gained an increased relevance. These platforms combine ease of programming and efficiency by providing fast, transactional access, to a transparently distributed shared state, and mechanisms aimed to elastically adjusting the resource consumption (nodes, memory, processing) in face of changes in the demand.

In these platforms, replication plays an essential role, given that it represents the primary mean to ensure data durability. The issue of transactional replication has been widely investigated in literature, targeting both classic databases [5] and transactional memory systems [4]. As a result, a large number of replication protocols have been proposed, based on significantly different design principles, such as, single-master vs multi-master management of update transactions [6], [7], lock-based vs atomic broadcast-based serialization of transactions [3], [5], optimistic vs pessimistic conflict detection schemes [4].

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

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

This paper addresses these issues by introducing MORPHR, a framework supporting automatic adaptation of the replication protocol employed by in-memory transactional platforms. The contributions of this paper are the following:

- We present the results of an extensive performance evaluation study using a popular open source transactional data grid (Infinispan by JBoss/Red Hat's), which we extended to support three different replication protocols, namely primary-backup [6], distributed locking based on two-phase commit [7], and atomic broadcast based certification [5]. We consider workloads originated by both synthetic and complex, standard benchmarks, and deployments over platforms of different scales. The results of our study highlight that none of these protocols can ensure optimal performance for all possible configurations, providing a strong argument to pursue the design of abstractions and mechanisms supporting the online reconfiguration of replication protocols.
- We introduce a framework, which we named MORPHR, formalizing a set of interfaces with precisely defined semantics, which need to be exposed (i.e. implemented) by an arbitrary replication protocol in order to support its online reconfiguration, i.e. switching to a different protocol. The proposed framework is designed to ensure both generality, by means of a protocol-agnostic generic reconfiguration protocol, and efficiency, whenever the cost of the transition between two specific replication protocols can be minimized by taking into account their intrinsic characteristics. We demonstrate the flexibility of proposed reconfiguration framework by showing that it can seamlessly encapsulate the three replication protocols mentioned above, via both protocol-agnostic and specialized protocol switching techniques.
- We validate the MORPHR framework, by integrating it in Infinispan, which allows to assess its practicality and efficiency in realistic transactional data grids. A noteworthy result highlighted by our experiments is that the MORPHR-

based version of Infinispan does not incur in perceivable performance overheads in absence of reconfigurations (which is expected to be the most frequent case), with respect to the non-reconfigurable version. We use this prototype to evaluate the latencies of generic and specialized reconfiguration techniques, demonstrating that the switching can be completed with a latency in the order of a few tens of milliseconds in a cluster of 10 nodes employing commodity-hardware.

• We show how to model the problem of determining the optimal replication protocol given the current operational conditions as a classification problem. By means of an experimental study based on heterogeneous workloads and platform scales, we demonstrate that this learning problem can be solved with high accuracy.

The remainder of the paper is structured as follows. Section II reports the results of a performance evaluation study highlighting the relevance of the addressed problem. The system architecture and its components are presented in Section III. The results of the experimental evaluation study are reported in Section V. Related work is analysed in Section VI. Section VII concludes the paper.

II. Motivations

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

A. Experimental Platform

We used a popular open-source in-memory transactional data grid, namely Infinispan [8] by Red Hat/JBoss, as reference platform for this study. At the time of writing, Infinispan is the reference NoSQL platform and clustering technology for JBoss AS, a mainstream open source J2EE application server. From a programming API perspective, Infinispan exposes a key-value store interface enriched with transactional support. Infinispan maintains data entirely in memory, using a weak-consistent (i.e. non-serializable) variant of multi-version concurrency algorithm to regulate local concurrency. Detection of remote conflicts, as well as data durability, are achieved by means of a Two Phase Commit [7] based (2PC) replication protocol. In order to assess the performance of alternative transaction replication protocols we have developed two custom prototypes of Infinispan, in which we replaced the native replication protocol with two alternative protocols, i.e. a Primary-Backup (PB) and Total Order Based (TOB) protocol. Note that due to the vastness of literature on transactional replication protocols, an exhaustive evaluation of all existing solutions is clearly infeasible. However, the three protocols that we consider, 2PC, PB, and TOB, represent different well-known archetypal approaches, which have inspired the design of a plethora of different variants in literature. Hence, we argue that they capture the key tradeoffs in most existing solutions. In the following we briefly overview these three protocols:

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

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

TOB: Similarly to 2PC, this protocol is a multi-master scheme that processes transactions without any synchronization during their execution phase. Unlike 2PC, however, the transaction serialization order is not determined by means of lock acquisition, but by relying on an Total Order Broadcast service (TOB) to establish a total order among committing transactions[9]. Upon their delivery by TOB, transactions are locally certified and either committed or aborted depending on the result of the certification. Being a lock-free algorithm, TOB does not suffer of the lock convoying phenomena in high contention scenarios. However, its AB-based commit phase imposes a larger communication overhead with respect to 2PC (and PB). This protocol has higher scalability potential than PB in write dominated workloads, but is also

	# Warehouses	% Order Status	% Payment	% New Order	# Threads
TW1	10	20	70	10	1
TW2	1	20	30	50	8
TW3	1	30	70	0	1

Table I
PARAMETERS SETTINGS FOR THE TPC-C WORKLOADS

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

Table II
PARAMETERS SETTINGS FOR THE RADARGUN WORKLOADS

more prone to incur in high abort rates in conflict intensive scenarios.

B. Benchmarks

We consider two popular benchmarks for transactional platforms, namely TPC-C and Radargun. The former is a standard benchmark for OLTP systems, which portrays the activities of a wholesale supplier and generates mixes of read-only and update transactions with strongly skewed access patterns and heterogeneous durations. We have developed an implementation of TPC-C that was adapted to execute on a NoSQL key/value store, which include three different transaction profiles: Order Status, a read-only long running transaction; New Order, a computation intensive update transaction that generates moderate contention; Payment, a short, conflict prone update transaction. Radargun, instead, is a benchmarking framework designed by JBoss to test the performance of distributed, transactional key-value stores. The workloads generated by Radargun are simpler and less diverse than those of TPC-C, but they have the advantage of being very easily tunable, thus allowing to easily explore a wide range of possible workload settings.

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

For Radargun we also consider three workloads, which we denote as RG1, RG2 and RG3. These three workloads are generated synthetically, and their characteristics can be controlled by tuning three parameters; the ratio of read-only vs update transactions; the number of read/write operations executed by (read-only/update) transactions; and the cardinality of the set of keys used to select, using a uniform distribution, the key accessed by each read/write operation. A detailed description of the parameter settings used for these workloads is reported in Table II.

C. Analysis of the results

We now report and discuss experimental data that illustrates the performance of 2PC, PB, and TOB protocols under different workloads. All the experiments reported in this paper have been obtained using a commodity cluster of 10 nodes. The machines are connected via a Gigabit switch and each one has Ubuntu 10.04 installed, 8 cores (2 Intel(R) Xeon(R) CPU E5506 @ 2.13GHz) and 32GB of RAM memory. We performed experiments with different cluster sizes. However, as the corresponding data sets show similar trends, for space constraints, for most workloads we only depict results in a 10 node cluster; the exception is the workload TW2, for which we also depict results in a 3 node cluster (denoted as TW2(3N)). The top plot in Figure 1 reports the throughput achieved by a each protocol normalized to the throughput of the best performing protocol (in the same scenario). The second and third plot from the top, on the other hand, report values on the transaction abort rate and commit latency.

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

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

III. ARCHITECTURAL OVERVIEW

The architecture of MORPHR is depicted in Figure 2. The system is composed by two macro components, a *Reconfigurable Replicated In-Memory Data Store* (RRITS), and a *Replication Protocol Selector Oracle* (RPSO).

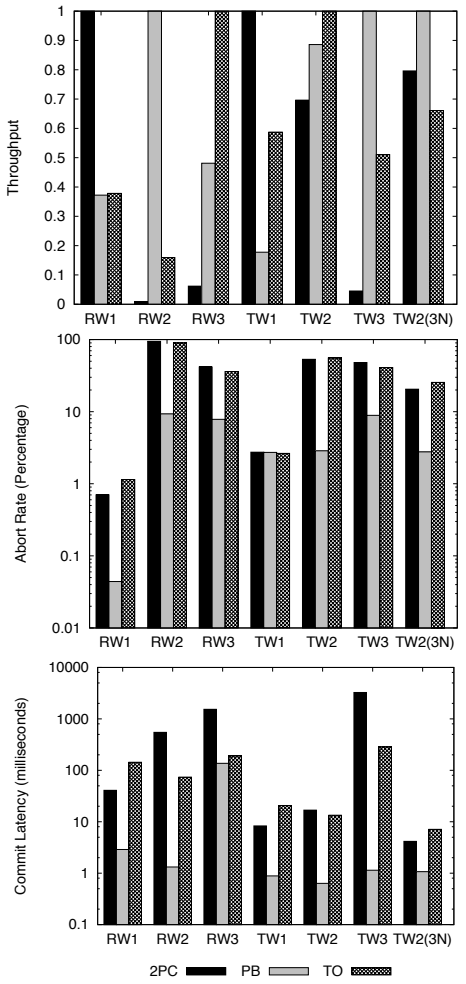

Figure 1. Comparing the performance of 2PC, PB and TOB protocols.

The RRITS externalizes user-level APIs for transactional data manipulation (such as those provided by a key/value store, as in our current prototype, or an STM platform), as well as APIs, used in MORPHR by the RPSO, that allow its remote monitoring and control (to trigger adaptation). Internally, the RRITS supports multi-layer reconfiguration techniques, which are encapsulated by abstract interfaces allowing to plug in, in a modular fashion, arbitrary protocols for replica coherency and concurrency control. A detailed description of this building block is provided in Section IV.

The RPSO is an abstraction that allows encapsulating different performance forecasting methodologies. The oracle implementation may be centralized or distributed. In a centralized implementation, the RPSO is a single process that runs in one of the replicas or in a separate machine. In the distributed implementation each replica has its own local instance of the oracle that coordinates with other instances to reach a common decision. In this work we chose to implement the centralized version, which will be discussed in more detail in Section IV-C.

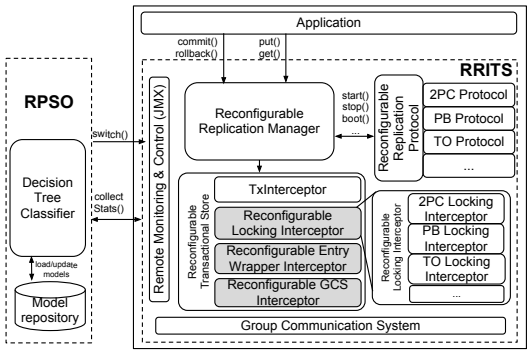

Figure 2. Architectural overview.

IV. RECONFIGURABLE REPLICATED IN-MEMORY TRANSACTIONAL STORE

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

A. Reconfigurable Replication Manager

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

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

MORPHR addresses this tradeoff by introducing a generic, protocol-agnostic reconfiguration protocol that guarantees the correct switching between two arbitrary replication protocols, as long as these adhere to a very simple interface (denoted as *ReconfigurableReplicationProtocol* in Figure 2). This interface allows MORPHR to properly control their execution (stop and start). In order to achieve full generality, i.e. to be able to ensure consistency in presence of transitions between any two replication protocols, MORPHR's generic reconfiguration protocol is based on a conservative "stop and go" approach, which enforces the termination of all transactions in the old protocol, putting the system in a quiescent state, before it starts executing the new protocol.

MORPHR requires that all its pluggable protocols implement the methods needed by this stop and go strategy (described below), benefiting extensibility and guaranteeing the generality of the approach. On the other hand, in order to maximize efficiency, for each pair of replication protocols (A,B), MORPHR allows an additional *protocol switcher*

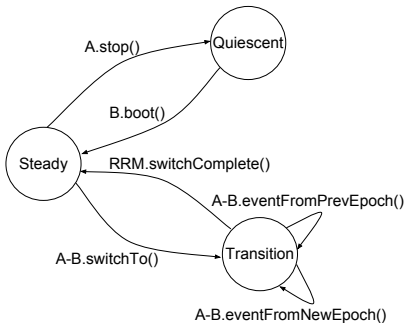

Figure 3. Finite state machine of the MORPHR reconfiguration schemes.

```
1  stop(boolean eager) {
2      block generation of new local transactions;
3      if eager then
4          abort any local executing transaction;
5      else
6          wait for completion of all local executing transactions;
7      end
8      broadcast (DONE);
9      wait received DONE from all processes;
10     wait for completion of all remote transactions;
11 }
```

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

algorithm to be registered, which interacts with the RRM via a well defined interface. The RRM uses such specialized reconfiguration protocols, whenever available, and otherwise resort to using the protocol-agnostic reconfiguration scheme.

Figure 3 depicts the state machine of the reconfiguration strategies supported by MORPHR. Initially the system is in the STEADY state, running a single protocol A. When a transition to another protocol B is requested, two paths are possible. The default path first puts the system in the QUIESCENT state and then starts protocol B which will put the system back to the STEADY state. The fast path consists of invoking the switching protocol. This protocol will place the system in a temporary TRANSITION state, where both protocol A and protocol B will coexist. When the switcher terminates, only protocol B will be executed and the system will be again in the STEADY state.

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

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

- *boot()*: This method is invoked to start the execution of a protocol from a QUIESCENT state, i.e., no transactions from the other protocol are active in the system, and implements any special initialization conditions required by the protocol.
- *stop*(*boolean* eager): This method is used to stop the execution of a protocol and putting the system in a QUIESCENT state. The protocol dependant implementation of this method must guarantee that, when it returns, there are no transactions active in the system executing with that protocol. The *eager* parameter is a boolean that allows to select if on-going transactions must be aborted immediately, or if the system should allow for on-going transactions to terminate in order to reach the QUIESCENT state.

The pseudo-code in Algorithm 1 provides an example implementation of this interface, for the case of the 2PC replication protocol described in Section II. First, all new local transactions are blocked. Then, the boolean received as input allows the programmer to decide whether to abort all locally executing transactions or allow them to complete their local and remote execution. When these finish executing, a DONE message is broadcast announcing that no more transactions will be issued by this replica in the current protocol. Before returning, it waits for this message from all the other replicas in the system and for the completion of any remote transactions executing in the that replica.

Fast switching reconfiguration: The default "stop and go" strategy ensures that, at any moment in time, no two transactions originated by different protocols can be concurrently active in the system. Non-blocking reconfiguration schemes avoid this limitation, by allowing some degree of overlap in the execution of protocols A and B during the reconfiguration. In order to establish an order on the reconfigurations, the RRM (i.e. each of the RRM instances maintained by the nodes in the system) relies on the notion of epochs. Each fast switching reconfiguration triggers a new epoch and all transaction events are tagged with the number of the epoch in which they were generated. Additionally, events generated from future epochs are buffered by the RRM, and dispatched to the corresponding protocol only when the local epochs are correctly aligned.

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

- *startSwitching()*: This method is invoked to initiate fast switching.
- *eventFromPrevEpoch*(transaction): This method processes a transaction that was initiated in a epoch previous to the one currently active in this replica.
- *eventFromNewEpoch*(transaction): This method processes a transaction that was initiated in a epoch subsequent to the one currently active in this replica.

Further, the RRM exposes a callback interface, via the *switchComplete*() method, which allows the protocol switcher to notify the ending of the transition phase of to the RRM, and which causes it to transit to the STEADY state. Like for the *stop()* method, a *protocol switcher* imple-

```
1  2PC-PB.startSwitching() {
2  |  broadcast (LOCALDONE);
3  }
4  2PC-PB.handleOldTx(event, tx) {
5  |  processEvent(event, tx);
6  }
7  2PC-PB.handleNewTx(event, tx) {
8  |  processEvent(event, tx);
9  }
10 upon received LOCALDONE from all nodes {
11 |  wait for completion of prepared remote 2PC txs;
12 |  // guarantee reconfiguration completion globally
13 |  broadcast (REMOTEDONE);
14 |  wait received REMOTEDONE from all nodes;
15 |  switchCompleted();
16 }
```

Algorithm 2: Fast Switching from 2PC to PB.

mentation must guarantee that when the *switchComplete*() method is invoked, every transaction active in the system is executing according to the final protocol.

In the following paragraphs we illustrate two examples of fast switching algorithms, for scenarios involving pairs of protocols whose concurrent coexistence raises different types of issues.

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

```
1  2PC-TOB.startSwitching() {
2  |  wait for local transactions in prepared state;
3  |  broadcast (LOCALDONE);
4  }
5  2PC-TOB.handleOldTx(event, tx) {
6  |  if event is of type Prepare then
7  |  |  rollback(tx);
8  |  end
9  |  processEvent(event, tx);
10 }
11 2PC-TOB.handleNewTx(event, tx) {
12 |  if tx conflicts with some tx' using 2PC then
13 |  |  wait for tx' to commit or abort;
14 |  end
15 |  processEvent(event, tx);
16 }
17 upon received LOCALDONE from all nodes {
18 |  // guarantee reconfiguration completion globally
19 |  broadcast (REMOTEDONE);
20 |  wait received REMOTEDONE from all nodes;
21 }
```

Algorithm 3: Fast switching from 2PC to TOB.

Fast switch from 2PC to TOB: 2PC and TOB protocols are radically different protocols, as they use different concurrency control schemes (lock-based vs lock-free) and communication primitives (plain vs totally ordered broadcast) that require the usage of incompatible data-structures/algorithms at the transactional data store level. Because of this, it is impossible for a node to start processing transaction with TOB if any locally generated 2PC transaction is still active. To this end, the fast switch implementation from 2PC to TOB, in Algorithm 3, returns from the *startSwitching* method (entering the new, TOB-regulated epoch) only after it has committed (or aborted) all its local transactions from the current epoch. During the TRANSITION state, a node replies negatively to any incoming prepare message for a remote 2PC transaction thus avoiding incompatibility issues with the currently executing TOB protocol. Transactions from the new TOB-based epoch, on the other hand, can be validated (and accordingly committed or aborted). However, if they conflict with any previously prepared but not committed 2PC transaction, the commit of the TOB transaction must be postponed until the outcome of previous 2PC transactions is known. Otherwise, it can be processed immediately according to the conventional TOB protocol. Analogously to the previous fast switching algorithm, also in this case a second global synchronization phase is required in order to ensure the semantics of the *switchCompleted* method.

B. Reconfigurable Transactional Store

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

concrete data store implementation in use.

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

The architectural diagram in Figure 2 illustrates how this principle was applied to one of the key elements of Infinispan, namely the interceptor chain that is responsible for capturing commands issued by the user and by the replication protocols and to redirecting them towards the modules managing specific subsystems of the data store (such as the locking system, the data container, or the group communication system). The interceptors whose behaviours had to be replaced due to an adaptation of the replication protocol, shown in gray in Figure 2, were replaced with generic reconfigurable interceptors, for which each replication protocol can provide its own specialized implementation. This allows to flexibly customize the behaviour of the data container depending on the replication protocol currently in use.

C. Replication Protocol Selector Oracle

As already mentioned, the Replication Protocol Selector Oracle component is a convenient form of encapsulating different performance forecasting techniques. In fact, different approaches, including analytical models [10] and machine-learning (ML) techniques [11], might be adopted to identify the replication protocol on the basis of the current operating conditions. In MORPHR we have opted for using ML-based forecasting techniques, as, thanks to their black-box nature, they can cope with arbitrary replication protocols, maximizing the generality and extensibility of the proposed approach.

The selection of the optimal replication protocol lends itself naturally to be cast as a *classification* problem [11], in which one is provided with a set of input variables (also called *features*) describing the current state of the system and is required to determine, as output, a value from a discrete domain (i.e., the best performing protocols among a finite set in our case). After preliminary experimentations with various ML tools (including SVM and Neural networks [11]), we have opted to integrate in MORPHR C5.0 [12], a state of the art decision-tree classifier. C5.0 builds a decision-tree classification model in an initial, off-line training phase during which a greedy heuristic is used to partition, at each level of the tree, the training dataset according to the input feature that maximizes information gain [12]. The output model is a decision-tree that closely classifies the training cases according to a compact (human-readable) rule-set, which can then be used to classify (decide the best performing replication strategy) future scenarios.

We shall discuss the methodology adopted in MORPHR to build ML-based performance models shortly, and focus for the moment on discussing how these models are used at runtime.

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

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

This approach results in an obvious tradeoff: the more the RPSO queries the ML, the faster it reacts to changes in the workloads, but it may happen that some are only momentary spikes and do not reflect a real change in the workload, thus triggering unnecessary changes in the configuration and preventing the system from achieving an optimal throughput. In our current prototype we use a simple approach based on a moving average over a window time of 30 seconds, which has proven successful with all the workloads we experimented with. Of course, the system may be made more robust by introducing techniques to filter out outliers [13], detect statistically relevant shifts of system's metrics [14], or predict future workload trends [15]. These are directions we plan to investigate in our future work, but that are out of the scope of this paper.

Construction of the ML model The accuracy achievable by any ML technique is well known [11] to be strictly dependant on the selection of appropriate input features. These should be, on one hand, sufficiently rich to allow the ML algorithm to infer rules capable of closely relating fluctuations of the input variables with shifts of the target variable. On the other hand, considering an excessively high number of features leads to an exponential growth of the training phase duration and to an increase of the risk of inferring erroneous/non-general relationships (a phenomenon called over-fitting [11]).

After conducting an extensive set of preliminary experiments, we decided to let MORPHR gather a base set of 14 metrics aimed to provide a detailed characterization of:

- the transactional workload: percentage of write transactions, number of read and write operations per read-only and

write transactions and their local and total execution time, abort rate, throughput, lock waiting time and hold time.
• the average utilization of the computational resources of each node and of the network: CPU, memory utilization and commit duration.

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

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

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

Finally, the last step of the model building phase consists in the execution of an automated feature selection algorithm, which is aimed at minimizing the risk of overfitting and maximizing the generality of the model by discarding features that are either too closely correlated among each other (and hence redundant), or too loosely correlated with the output variable (and hence useless). Specifically, we rely on the Forward Selection [16] technique, a greedy heuristic that progressively extends the set of selected features till the accuracy it achieves when using 10-fold cross-validation on the training set is maximized.

V. Evaluation

This section presents the results of an experimental study aimed at assessing three main aspects: i) the accuracy of the ML-based selection of the best-fitting replication protocol (see Section V-A); ii) the efficiency of the alternative protocol switching strategies discussed in Section IV (see Section V-B); iii) the overheads introduced by the online monitoring and re-configuration supports employed by MorphR to achieve self-tuning (see Section V-B).

A. Accuracy of the RPSO

In order to assess the accuracy of the RPSO we generated approx. 75 workloads for both the TPC-C and Radar Gun, varying uniformly the parameters that control the composition of transaction mixes and their duration. More in particular, for each of the 3 TPC-C workloads described in Section II, we explored 25 different configuration of the percentages of Order Status, Payment and New Order Transactions. Analogously, starting from each of the 3 Radargun workloads reported in Table I, we explored 27 different variations of the parameters that control the percentage of write transactions, and the number of read/write operations both in read-only and update transaction. This workload generation strategy allowed us to obtain a balanced data containing approximately the same number of workloads for which each of the three considered protocols results to be the optimal choice.

We run each of the above workloads with the 3 considered protocols, yielding a data set of approximately 1350 cases that serves as the basis for this study. The continuous lines in Figure 4 provide an interesting perspective on our data set, reporting the normalized performance (i.e., committed transactions per second) of the 2nd and 3rd best choice with respect to the optimal protocol for each of the considered workload.

The plots highlight that, when considering the Radargun workloads, the selection of the correct protocol configuration has a striking effect on system's throughput: in 50% of the workload, the selection of the 2nd best performing protocol is at least twice slower than the optimal protocol; further, the performance grows to at least 10x in case the worst performing protocol were to be erroneously selected by the RPSO. On the other hand, the TPC-C workload shows less dramatic, albeit still significant, differences in the relative performances of the protocols. Being the performance of the three protocols relatively closer with TPC-C than with Radargun, one may argue that the classification problem at hand is harder in the TPC-C scenario, at least provided that one evaluates the accuracy of the ML-based classifier exclusively in terms of misclassification rate. On the other hand, in practical settings, the actual relevance of a misclassification is clearly dependant on the actual throughput

Benchmark	W/O Prot.	With Prot.	Three Models
Radargun	1.68%	1.68%	1.25%
TPC-C	8.17%	5.44%	4.58%

Table III
PERCENTAGE OF MISCLASSIFICATION.

loss to the sub-optimal protocol selection. In this sense, the Radargun's workloads are significantly more challenging than TPC-C's ones. Hence, whenever possible, we will evaluate the quality of our ML-based classifiers using both metrics, i.e. misclassification rate and throughput loss vs optimal protocol.

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

Table III shows the percentage of misclassification for each case. These results were obtained by averaging the results of ten models, each one built using ten-fold cross validation. The results show that, especially for the TPC-C benchmark (which, as discussed above, entails scenarios that are more challenging to classify than when using Radargun), the misclassification rate can be significantly lowered by incorporating in the model information on the protocol in use when characterizing the current workload and system's state. In particular, using distinct models for each protocol, as expected, we minimize the chances that the ML is misled by the simultaneous presence of training cases exhibiting similar values for the same feature but associated with different optimal protocols (because measured when running different protocols), or, vice versa, of training cases associated with the same optimal protocol but exhibiting different values for the same feature (again, because observed using different protocols). At the light of this result, in MORPHR, as well as in the remainder of this section, we opted for using a distinct model for each protocol.

The dashed lines reported in Figure 4 allow us to evaluate the accuracy of the RPSO from a different perspective, reporting the cumulative distribution of the throughput achievable by following the RPSO's predictions for each workload, normalized to the throughput achieved by the optimal protocol for that workload. In order to assess the extrapolation power of the classifiers built using the proposed methodology we progressively reduce the training set from 90% to 40% of the entire data set, and use, the remaining cases as test sets.

Both benchmarks show the same trend. As the training set becomes larger, the percentage of cases with sub-optimal throughput decreases. Furthermore, in these cases, the loss throughput in absolute value, when compared to the optimal choice, also decreases. In fact, it can be observed that even for models built using the smallest training set, and considering the most challenging benchmark (namely TPC-C), the performance penalty w.r.t. the optimal configuration is lower than 10% in around 85% of the workloads. On the other hand, for models built using the largest training set, the throughput penalty is lower than 10% in about 90% of the cases. Table IV presents information concerning the throughput loss averaged across all workload when using ML-based models, again built using training sets of different sizes. The table also reports the average throughput loss of the 2nd and 3rd best performing protocol for each scenario, as well as the average performance penalty that one would achieve using a trivial random selection strategy.

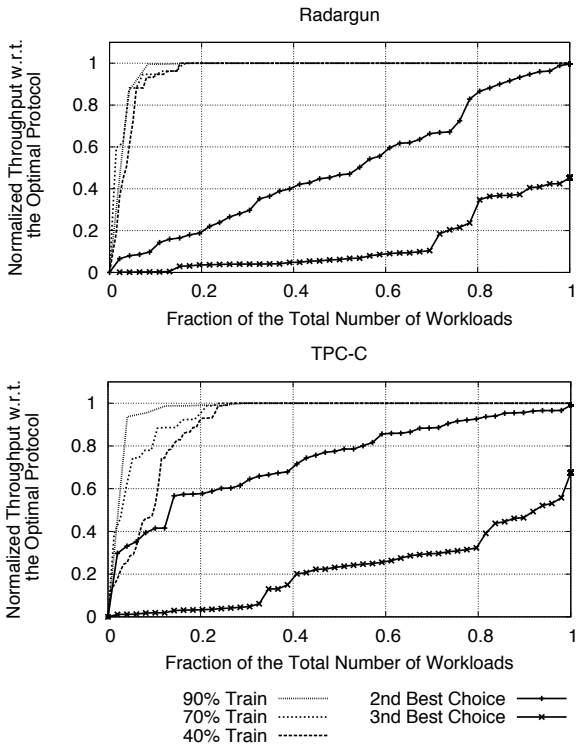

Figure 4. Cumulative distribution of the normalized throughput vs the optimal protocol

Benchmark	90% train	70% train	40% train	2nd prot.	3rd prot.	Random choice
Radargun	99%	98%	96%	86%	49%	55%
TPC-C	99%	96%	91%	77%	25%	65%

Table IV
AVERAGE PERCENTAGE OF THROUGHPUT LOSS.

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

B. *Fast switch* vs *Default Switch*

We now analyse the performance of the specialized fast switching algorithms, contrasting it with that achievable by the generic, but less efficient, stop and go switching approach. For this purpose we built a simple synthetic benchmark designed to generate transactions having a user-

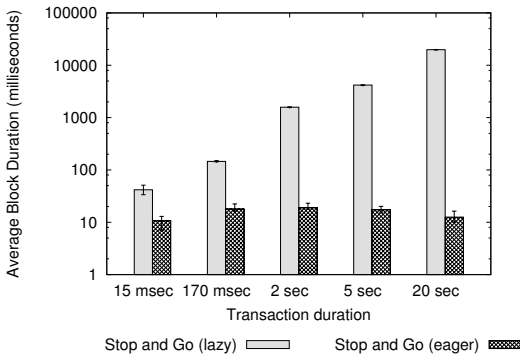

Figure 5. Blocking time during the switch between 2PC and PB

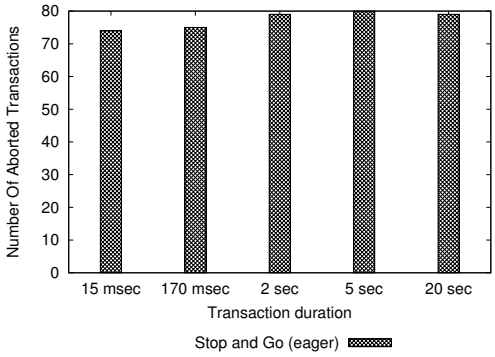

Figure 6. Aborted transactions during the switch between 2PC and PB

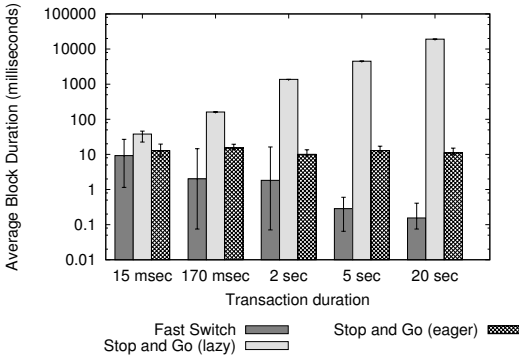

Figure 7. Blocking time during the switch between 2PC and TOB

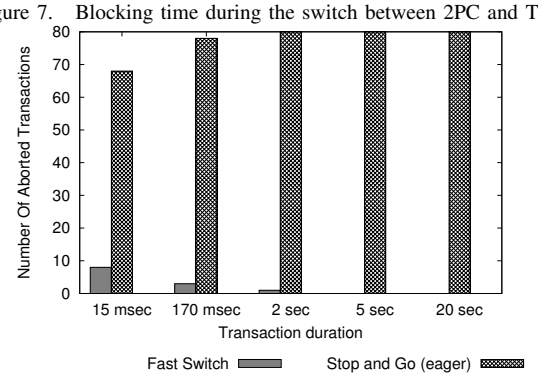

Figure 8. Aborted transactions during the switch between 2PC and TOB

tunable duration, which we let vary from 15 milliseconds to 20 seconds. Furthermore, we have experimented with different fast switching algorithms and both with the eager and the lazy versions of the stop and go algorithm (recall that with the eager version ongoing transactions are simply aborted, whereas with the lazy version we wait for pending transactions to terminate before switching).

We start by considering the fastest switching specialized algorithm we designed, namely the one that commutes from 2PC to PB (Alg. 2). Figure 5 shows the average blocking time, i.e., the period during which new transactions are not accepted in the system due to the switch (naturally, the shorter this period the better). Figure 6 presents the abort during the switching process. The figures show values for the fast switching algorithm, and for both the lazy and eager version of stop and go.

As previously discussed, the fast switching algorithm has no blocking phase and for the scenarios where the duration of transactions is larger, this can be a huge advantage when compared with the lazy stop and go approach. As expected, the fast switching algorithm is independent of the transaction duration, as it is not required to abort or to wait for the termination of transaction started with 2PC before accepting transaction with PB. On the other hand, the lazy stop and go approach, while avoid aborting transactions, can introduce a long blocking time (which, naturally, gets worse for scenarios where transactions have longer duration). In conclusion, the eager stop and go trades a lower stopping time for an high abort rate.

Let us now consider the fast switching algorithms for commuting from 2PC to TOB (Alg. 3), whose performance is evaluated in the Figures 7 and 8. In this fast switch algorithm nodes must first wait for all local pending transactions initiated with 2PC to terminate before accepting transactions to be processed by TOB. Therefore, this algorithm also introduces some amount of blocking time that, although smaller than in the case of the stop and go switching algorithm, is no longer negligible. Nevertheless, the advantages of fast switching are still very significant when transactions are very long.

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

C. *Performance of* MORPHR

Figure 9 compares the throughput of MORPHR with that of a statically configured, non-adaptive Infinispan. The model was trained with the TPC-C dataset previously presented, from which we removed the workloads TW1-3

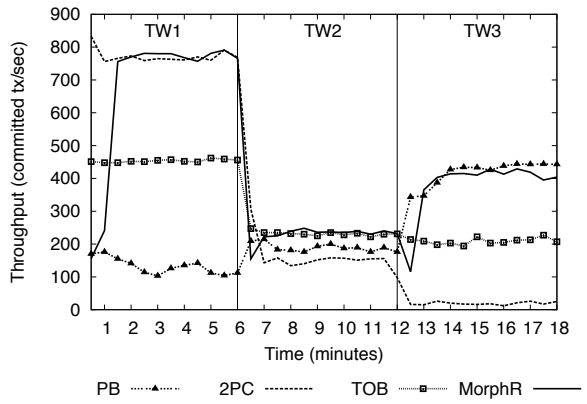

Figure 9. Comparison of the performance of MORPHR with static configurations

reported in Table I, which we inject in the system for a duration of 6 minutes. We configured the RPSO to query the system state every 30 seconds and predict the protocol to be used. The plots show that, whenever the workload changes, the RPSO detects it and promptly switches to the most appropriate protocol. As expected, the performance of MORPHR keeps up with that of the best static configuration. We can also observe that the overheads introduced by the supports for adaptivity are very reduced given that, when MORPHR stabilizes, its throughput is very close to the static configuration. For these experiments we placed the RPSO in one of the nodes running MORPHR as this is a very lightweight component. More specifically, each query made to model takes on average approx. 50 μseconds.

VI. RELATED WORK

We classify related work into the following categories: i) work on protocol reconfiguration in general; ii) work on automated resource provisioning; iii) work on self-tuning in databases systems, both distributed and centralized; iv) and work on adaptive STMs. We will address each of these categories in turn.

An extensive set of works has been produced on dynamic protocol reconfiguration [17], [18], [19]. A large part of this work has focused on the reconfiguration of communication protocols. For instance,[19] proposes an Atomic Broadcast (AB) generic meta-protocol that allows to stop an executing instance of an AB protocol, and to activate a new one. This problem is inherently related to the problem of adapting at run-time the replication scheme of a transactional system. In the latter case, which represents the focus of our work, adaptation encompasses a larger stack of software layers, and it is necessary to take into account the additional complexities due to the inter-dependencies between the replica control (keeping the copies consistent) and concurrency control (ensuring isolation in presence of concurrent data accesses) schemes. Also, in MORPHR we address also the issue of how to automatically determine *when* to trigger adaptation, and not only *how*.

The area of automated resource provisioning is related to this work as it aims at reacting to changes in the workload and access patterns to autonomically adapt the system's resources. Examples include works in both transactional[20], [10] and non-transactional application domains, such as Map-Reduce[21] and VM sizing[22]. Analogously to MORPHR, several of these systems also use machine-learning techniques to drive the adaptation. However, the problem of reconfiguring the replication protocol raises additional challenges, e.g. by demanding dedicated schemes to enforce consistency during the transitioning between two replication strategies.

To the best of our knowledge, the work in[23] pioneers the issues associated with adaptation in transactional systems (specifically, DBMSs). In particular, this paper identifies a set of sufficient conditions for supporting non-blocking switches between concurrency control protocols. While the identification of these conditions has interesting theoretical implications, unfortunately, in order to satisfy them it is necessary to enforce very stringent assumptions (such as knowing a-priori whether the transactions regulated by two simultaneously executing protocols will exhibit any data dependency), which restricts significantly the practical viability of this approach. Our solution, on the other hand, relies on a framework that supports switching between generic replication protocols without requiring any assumption on the workload generated by applications. Several other approaches have also been proposed based on the idea to automatically analyse the incoming workload, e.g.[24], to automatically identify the optimal database physical design or self-tune some of the inner management schemes, e.g.[25]. However, none of these approaches investigated the issues related to adapt the replication scheme. Even though the work in[26] presents a meta-protocol for switching between replication schemes, it does not provide a mechanism to autonomically determine the most appropriate scheme for the current conditions.

Finally, a number of works have been aimed at automatically tuning the performance of Software Transactional Memory (STM) systems, even if most of these works dos not consider replicated systems. In[27], the authors present a framework for automatically tuning the performance of the system by switching between different STM algorithms. This work was based in RSTM[28], which allows changing both STM algorithms and configuration parameters within the same algorithm. The main difference between RSTM and our work is that the later system must stop processing transactions whenever changing the (local) concurrency control algorithm, whereas MORPHR provides mechanisms allowing the coexistence of protocols while the switch is in process. The works in [29] and [30] also allow changing configuration parameters, but in our framework we only consider changing the protocol as a whole.

PolyCert [34] uses ML techniques to determine which is the most appropriate replication protocol according to

each transaction's characteristics for in-memory transactional data grids. However, in Polycert it is only possible to use protocols from the same family, namely certification based replication protocols, which only differ in the way transactions are validated. In this work, we address the more complex and generic problem of adaptive reconfiguration among arbitrary replication schemes.

VII. Conclusion

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

References

[1] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *Proc. VLDB 2007*. VLDB Endowment, 2007, pp. 1150–1160.

[2] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, "Cloud-TM: Harnessing the cloud with distributed transactional memories," *SIGOPS Operating Systems Review*, vol. 44, pp. 1–6, 2010.

[3] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *Proc. PPoPP 2006*. New York, NY, USA: ACM, 2006, pp. 198–208.

[4] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proc. PRDC 2009*, Shanghai, China, 2009, pp. 307–313.

[5] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

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

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

[8] F. Marchioni, *Infinispan Data Grid Platform*. Packt Publishing, Limited, 2012.

[9] X. Defago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.

[10] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional auto scaler: Elastic scaling of in-memory transactional data grids," in *Proc. ICAC 2012*, 2012.

[11] T. M. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997.

[12] J. R. Quinlan, "C5.0," http://www.rulequest.com/see5-info.html.

[13] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artif. Intell. Rev.*, vol. 22, no. 2, pp. 85–126, Oct. 2004.

[14] E. Page, "Continuous inspection schemes," *Biometrika*, pp. 100–115, 1954.

[15] R. Kalman *et al.*, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[16] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[17] X. Liu, R. V. Renesse, M. Bickford, C. Kreitz, and R. Constable, "Protocol switching: Exploiting meta-properties," in *Proc.WARGC 2001*, 2001, pp. 37–42.

[18] W. Chen, M. Hiltunen, and R. Schlichting, "Constructing adaptive software in distributed systems," in *Proc. ICDCS 2001*. IEEE Computer Society, 2001, pp. 635–643.

[19] O. Rütti, P. T. Wojciechowski, and A. Schiper, "Structural and algorithmic issues of dynamic protocol update," in *Proc. IPDPS 2006*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 133–133.

[20] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ, "Activesla: a profit-oriented admission control framework for database-as-a-service providers," in *Proc. SOCC 2011*. New York, NY, USA: ACM, 2011, pp. 15:1–15:14.

[21] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," in *Proc. SOCC 2011*. New York, NY, USA: ACM, 2011, pp. 18:1–18:14.

[22] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. A. B. Fortes, "Fuzzy modeling based resource management for virtualized database systems," in *Proc. MASCOTS 2011*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 32–42.

[23] B. Bhargava and J. Riedl, "A model for adaptable systems for transaction processing," *IEEE Trans. on Knowl. and Data Eng.*, vol. 1, no. 4, pp. 433–449, Dec. 1989.

[24] P. Martin, S. Elnaffar, and T. Wasserman, "Workload models for autonomic database management systems," in *Proc. ICAS 2006*. Washington, DC, USA: IEEE Computer Society, 2006, p. 10.

[25] N. Bruno and S. Chaudhuri, "An approach to physical design tuning," in *Proc. ICDE 2007*, 2007, pp. 826–835.

[26] M. I. Ruiz-Fuertes and F. D. Munoz-Escoi, "Performance evaluation of a metaprotocol for database replication adaptability," in *Proc. SRDS 2009*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 32–38.

[27] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear, "A transactional memory with automatic performance tuning," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 54:1–54:23, Jan. 2012.

[28] M. F. Spear, "Lightweight, robust adaptivity for software transactional memory," in *Proc. SPAA 2010*. New York, NY, USA: ACM, 2010, pp. 273–283.

[29] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proc. PPoPP 2008*. New York, NY, USA: ACM, 2008, pp. 237–246.

[30] V. Marathe, W. Scherer, and M. Scott, "Adaptive software transactional memory," in *Distributed Computing*, ser. Lecture Notes in Computer Science, P. Fraigniaud, Ed. Springer Berlin / Heidelberg, 2005, vol. 3724, pp. 354–368.

[31] M. Holanda, A. Brayner, and S. Fialho, "Introducing self-adaptability into transaction processing," in *Proc. SAC 2008*. New York, NY, USA: ACM, 2008, pp. 992–997.

[32] M. Castro, L. Goes, C. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *Proc. HiPC 2011*, dec. 2011, pp. 1 –10.

[33] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "Autonomic resource management in virtualized data centers using fuzzy logic-based approaches," *Cluster Computing*, vol. 11, no. 3, pp. 213–227, 2008.

[34] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: Polymorphic self-optimizing replication for in-memory transactional grids," in *Proc. Middleware 2011*.