State Machine Replication for the Masses with BFT-SMaRt

Alysson Bessani, João Sousa and Eduardo Alchieri

DI–FCUL–TR–2013–07

DOI:10455/6897

(http://hdl.handle.net/10455/6897)

December 2013

State Machine Replication for the Masses with BFT-SMaRt

Alysson Bessani[1], João Sousa[1], Eduardo Alchieri[2]
[1] University of Lisbon, FCUL/LaSIGE – Portugal, [2] University of Brasilia – Brazil

Abstract

The last fifteen years has seen an impressive amount of work on protocols for Byzantine fault-tolerant (BFT) state machine replication. However, there is still a lack of practical and reliable software libraries implementing this technique. BFT-SMaRt is an open-source Java-based library implementing robust BFT state machine replication. Some of the key features of this library that distinguishes it from similar works (e.g., PBFT and UpRight) are improved reliability, modularity as a first-class property, multicore-awareness, reconfiguration support and an flexible API. When compared with other SMR libraries, BFT-SMaRt achieves better performance and is able to deal with a number of real-world faults that previous implementations can not.

1 Introduction

The last decade and a half has seen an impressive amount of papers on Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) (e.g., [3, 13, 16, 21–23, 31, 32], to cite just a few), but almost no practical use of these techniques in real deployments.

Our view of this situation is that the fact that there are no robust-enough implementations of BFT SMR available, only prototypes used for validating novel ideas in papers, makes it quite difficult to use this kind of technique. The general perception is that implementing BFT protocols is too complex and that commission faults are rare and can be normally dealt with simpler techniques like checksums [17].

To the best of our knowledge, from all "BFT systems" that appeared on the fifteen years, only the early PBFT [13] and the more recent UpRight [15] implement a complete replication system. However, PBFT employs a single-threaded architecture which does not fully exploit modern hardware and UpRight uses two additional layers of servers between clients and replicas, besides presenting a performance an order of magnitude lower than the other systems. Furthermore, both PBFT and UpRight are plagued by bugs and are not maintained anymore. Even considering crash-only fault-tolerant (CFT) replication libraries, which are usually based on the many variants of the Paxos algorithm [24], it seems there is still no widely-used robust implementation that can be used for developing dependable services. As a result, every organization that requires such services need to develop its own implementation (e.g., [14]).

In this paper we describe our effort in implementing (and maintaining) BFT-SMaRt [1], a robust Java-based BFT SMR library which implements a protocol similar to the one used in PBFT. BFT-SMaRt targets not only high-performance in fault-free executions, but also correctness if faulty replicas exhibit arbitrary behavior. Besides its robustness, BFT-SMaRt is the first BFT SMR system to provide efficient and transparent support for durable services [9] and to fully support reconfiguration of the replica set [4, 25].

The main contribution of this paper is to fill a gap in the BFT literature by documenting the implementation of this kind of system, including associate protocols for state transfer and reconfiguration. Additionally, the paper presents an extensive evaluation of BFT-SMaRt, comparing it with previous systems and shedding light on some performance tradeoffs related with the tolerance of crashes vs. Byzantine faults.

The paper is organized as follows: §2 and §3 describe the design of BFT-SMaRt and its implementation, respectively. §4 gives an overview of the library's API and programing model. §5 presents alternative configurations for BFT-SMaRt. §6 describes an extensive evaluation of our system. §7 highlights some lessons learned during the development and maintenance of the system. Finally, §8 presents our concluding remarks.

2 BFT-SMaRt Design

The development of BFT-SMaRt started at the beginning of 2007 to implement a BFT total order multicast protocol for the replication layer of the DepSpace coordination service [8]. In 2009 we revamped this implementation to make it a complete BFT replication library, including features such as checkpoints and state transfer. Nonetheless, it was only during the TClouds project[1] (2010-2013) that we improved the system substantially in terms of functionality and robustness.

[1]`https://www.tclouds-project.eu`.

2.1 Design Principles

BFT-SMaRt was developed with the following design principles in mind:

Tunable fault model. By default, BFT-SMaRt tolerates *non-malicious Byzantine faults*, a realistic (albeit pessimistic) system model in which messages can be delayed, dropped and even corrupted, while processes can crash or have their state and code corrupted, taking any spurious action as a consequence. All these behaviors have been observed in real systems and components (see [17] for an overview). We believe this is an appropriate fault model for a pragmatical system to support the implementation critical services. Besides that, BFT-SMaRt also supports the use of cryptographic signatures for improved tolerance to *malicious Byzantine faults*, or the use of a simplified protocol, similar to Paxos [24], to tolerate only crashes and message corruptions.[2]

Simplicity. The emphasis on protocol correctness lead us to avoid the use of optimizations that could bring extra complexity both in terms of deployment and coding or add corner cases to the system. For this reason, we avoid techniques that, although promising in terms of performance (e.g., speculation [23] and pipelining [21]) or resource efficiency (e.g., trusted components [22, 32] or IP multicast [13, 23]), would make our code more complex to make correct (due to new corner cases) or deploy (due to lack of infrastructure support). This emphasis also made us chose Java instead of C/C++ as the implementation language. In §6 we show that even with these choices, the performance of BFT-SMaRt is similar or better than some of these optimized SMR implementations.

Modularity. BFT-SMaRt implements the Mod-SMaRt protocol, a *modular* SMR protocol that uses a well defined consensus module in its core [30]. On the other hand, systems like PBFT implement a *monolithic* protocol where the consensus algorithm is embedded inside of the SMR, without a clear separation. While both protocols are equivalent at run-time, modular alternatives tend to be easier to implement and reason about, when compared to monolithic protocols. Besides the existence of modules for reliable communication, client requests ordering and consensus, BFT-SMaRt also implements state transfer and reconfiguration modules, which are completely separated from the agreement protocol, as show in Figure 1.

Simple and Extensible API. Our library encapsulates all the complexity of SMR inside a simple and extensible API that can be used by programmers to implement deterministic services. More precisely, if the service strictly follows the SMR programming model, clients can

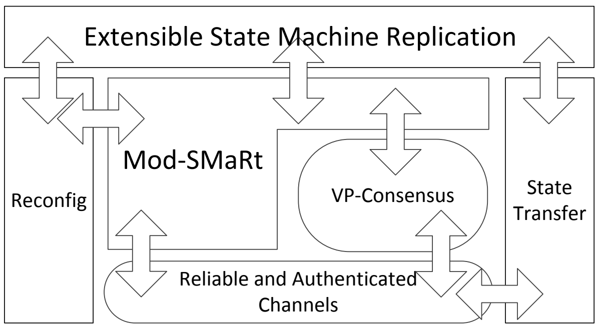

Extensible State Machine Replication
Mod-SMaRt
Reconfig
VP-Consensus
State Transfer
Reliable and Authenticated Channels

Figure 1: The modularity of BFT-SMaRt.

use a simple *invoke(command)* method to send commands to the replicas, that implement an *execute(command)* method to process the command, after it is totally ordered by the framework. If the application requires advanced features not supported by such basic programming model, these features can be implemented with a set of alternative calls, callbacks or plug-ins both at client- and server-side (e.g., custom voting by the client, reply management and state management, among others).

Multi-core awareness. BFT-SMaRt takes advantage of ubiquitous multicore architecture of servers to improve some costly processing tasks on the critical path of the protocol. In particular, we make our system throughput scale with the number of hardware threads supported by the replicas, specially when signatures are enabled and more computing power is needed for their verification.

2.2 System Model

BFT-SMaRt assumes a dynamic distributed system in which a universe of processes can be divided in two non-intersecting subsets: replicas and clients. Each process of the system has a unique identifier. During a dynamic system execution, a sequence of views is installed to denote the reconfigurations due to replicas joins and leaves. A view is composed by a set of replicas identifiers.

At any real time t, only replicas in the *current view* cv of the system are considered by the BFT-SMaRt protocols. The list of servers in cv represents the most up-to-date view installed in the system. We denote by $cv.n$ the number of replicas in cv and $cv.f < cv.n/3$ the number of replicas in cv allowed to fail arbitrarily. When reconfigurations are not being considered we suppress the cv prefix. We assume replicas can crash (or faulty replicas can be shutdown) and later recover. Moreover, an unbounded number of clients can also fail arbitrarily.

BFT-SMaRt requires an eventually synchronous system model for ensuring liveness, like other SMR protocols [13, 24], and reliable authenticated point-to-point links between processes for communication. These links are implemented using message authentication codes

[2]Unless stated otherwise, we focus on the BFT setup of the system. The crash fault tolerance is discussed later in §5.

(MACs) over TCP/IP. The symmetric keys for channels between replicas are generated at runtime using Signed Diffie-Hellman, which requires every replica to have a pair of (public and private) keys. The keys for client-replica channels are generated based on the ids of the endpoints, without the need for clients to have key pairs. Notice this is in accordance with our assumption of non-malicious Byzantine faults. Furthermore, strong authentication of clients is still available if signed requests are enabled (see §5.2).

2.3 Core Protocols

BFT-SMaRt uses a number of protocols for implementing state machine replication. In this section we give a brief overview of these protocols and refer the interested reader to the papers describing them [9, 12, 30].

2.3.1 Total Order Multicast

Total order multicast is achieved using the Mod-SMaRt protocol [30] together with the Byzantine consensus algorithm described in [12]. Clients send their requests to all replicas in *cv*, and wait for their replies.[3] In the absence of faults and presence of synchrony, BFT-SMaRt executes in *normal phase*, whose message pattern is illustrated in Figure 2. This phase considers the execution of a sequence of consensus instances, each of them deciding the order of a batch of one or more client requests. Each consensus execution *i* begins with one of the replicas designated as the leader (initially the replica with the lowest id) proposing some value for the consensus through a *PROPOSE* message. All replicas that receive this message verify if its sender is the current leader, and if the value proposed is valid (i.e., it contains only authenticated requests not yet ordered), they weakly accept the value being proposed, sending a *WRITE* message to other replicas. If some replica receives more than $\frac{n+f}{2}$ *WRITE* messages for the same value, it strongly accepts this value and sends an *ACCEPT* message to other replicas. If some replica receives more than $\frac{n+f}{2}$ *ACCEPT* messages for the same value, this value is used as the decision for consensus. The *ACCEPT* messages of a consensus instance form a certificate its decision. Therefore, such messages include a MAC vector, to enable the validation of a decision after a leader change [30]. Finally, the decision is logged (either in memory or disk) and the requests in the decided batch are executed in a deterministic order.

The normal phase of the protocol is executed in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, the *synchronization phase* might be triggered. During this phase, Mod-SMaRt must ensure three things: (1) a quorum of $n-f$ replicas must have the pending messages that caused the timeouts; (2) correct replicas must exchange logs to converge to the same consensus instance; and (3) a timeout is triggered in this consensus, proposing the same leader at all correct replicas (see [30] for details).

[3]See §2.3.3 for the discussion of how clients obtain and maintain an up-to-date *cv*.

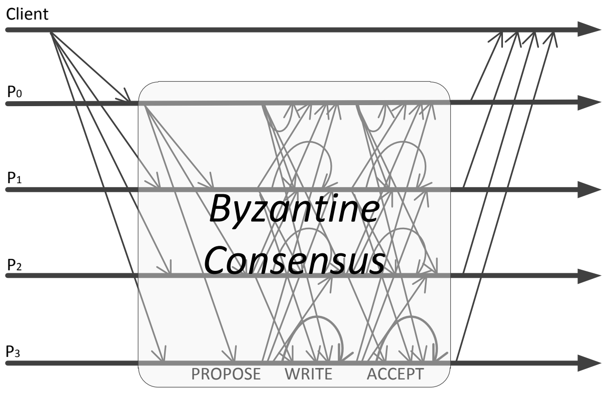


Figure 2: BFT-SMaRt normal phase message pattern.

2.3.2 State Transfer

In order to implement a practical state machine replication, the replicas should be able to be repaired and reintegrated in the system, without restarting the whole replicated service. Furthermore, the possibility of correlated failures that can bring down more than f replicas of the system at once, requires the employment of durability techniques (e.g., the use of stable storage) to be able to recover the whole system in such situations. BFT-SMaRt implements the efficient durability techniques described in [9] to deal with the recovery of replicas or the whole system. In the following we give an overview of such techniques.

By default, replicas store each batch of ordered requests to a (stable) log and, periodically, take snapshots of the application state and store it in stable memory. These two techniques incur a non-negligible performance penalty when disks are used. To mitigate this effect, the logging of operations can be done in batches and in parallel with their execution while snapshots are taken at different points of the execution in different replicas [9]. This behavior is implemented in an well-defined layer between the replication protocol and the application, and it can be changed in accordance with the requirements of the application.

The default state transfer protocol can be triggered either when (1) a replica crashes but is later restarted, (2) a replica detects that it is slower that the others, (3) a synchronization phase is triggered but the log is truncated beyond the point at which the replica could apply the operations, and (4) a replica is added to the system while it is running (see next section). When any of these scenarios are detected, the replica sends a *STATE_REQUEST* message to all the other replicas asking for the application's state. Upon receiving this request, they reply with a *STATE_REPLY* message containing the version of the state that was requested by the replica. Instead of having one replica sending the complete state (checkpoint and log) and others sending cryptographic hashes for validating this state, as is done in PBFT and other systems, we use a partitioning scheme in which one replica sends a checkpoint and the others send parts of the logs [9].

2.3.3 Reconfiguration

All previous BFT SMR systems assume a static system that cannot grow or shrink over time. BFT-SMaRt, on the other hand, provides an additional protocol that enables the system current view cv to be modified at runtime, i.e., replicas can be added or removed without stopping the system. In order to accomplish this, BFT-SMaRt uses a special type of client named *View Manager*, which is a trusted third party managed only by system administrators. It can also remain off-line, being required only for adding and removing replicas.

Server operation. The reconfiguration protocol works as follows: the View Manager issues a *signed request* containing a special reconfigure operation to be processed by the Mod-SMaRt algorithm just like any other client operation. Through this operation, the View Manager notifies the system about the IP addresses, ports and ids of the replicas it wants to add to (or remove from) the system. In the current BFT-SMaRt implementation, the View Manager can also request the update of the number of failures tolerated in the system ($cv.f$, which is also part of a view). Since these operations are totally ordered (just like client requests), all correct replicas will adopt the same view as the system' current view cv at any given point in the execution of client operations.

A subtle issue with reconfiguration requests is that *ACCEPT* messages exchanged in the consensus in which they are ordered should be signed (instead authenticated using MAC vectors). This happens because such messages are used to build certificates that may be needed in a future synchronization phase (i.e., leader change), and MAC vectors generated in a view cannot always be verified in a posterior view.

Once the View Manager operation is ordered, it is not delivered to the application. Instead, the request signature is verified to assess if it was produced using the view manager private key. If the signature is valid, the system current view cv is updated in accordance with the updates requested in the reconfigure operation. Moreover, the replicas start establishing a secure channel with the new replicas joining the system (or closing channels with the replicas leaving the system). Finally, the replicas reply to the View Manager informing it if the view change succeeded. If so, the View Manager sends a special message to the replicas that are waiting to be added to the system, informing them that they can start executing in cv. After this point, the joining replicas trigger the state transfer protocol to bring themselves up to date.

Client operation. In order to support reconfigurations, each client c also needs to handle a current view variable cv_c that stores the current view known by itself. All client operations need to carry cv_c and the replicas reject any operation issued to an old view, replying instead with their current view cv. The client then updates cv_c and restarts its operation, avoiding access to an outdated view.

Like several other reconfigurable systems [4], to ensure that a slow client c always terminate its operation op, the number of reconfigurations executed concurrently with op must to be finite. This ensures that c will restart op due to reconfigurations a finite number of times, eventually completing it.

The last requirement of a reconfigurable system is that, before accessing the system, a client must obtain the system' current view. This can be done with the use of a directory service [4, 26], for example. The current BFT-SMaRt implementation supplies interfaces for the programmers be able to implement their own view repository.

3 Implementation

The codebase of BFT-SMaRt contains less than 13.5K lines of commented Java code distributed in little more than 90 classes and interfaces. This is significantly less than what was used in similar systems: PBFT [13] contains 20K lines of C code and UpRight [15] contains 22K lines of Java code. Even JPaxos [28], the most complete open-source CFT replication library we are aware of, has more than 22K lines of commented Java code.

3.1 Building blocks

To achieve modularity, we defined a set of building blocks (or modules) containing the core functionality of BFT-SMaRt. These blocks are divided in three groups: *communication system*, *state machine replication* and *state management*. The first encapsulates everything related to client-to-replica and replica-to-replica communication, including authentication, replay attacks detection, and (re)establishment of communication channels after a failure or reconfiguration. The second implements the core algorithms for establishing total order of requests. The third deals with state management and is described in [9].

3.1.1 Communication system

The communication system encapsulates all the code required for receiving requests from clients and messages from other replicas, and sending messages to other processes addressed by their numeric ids. The three main modules are:

- **Client Communication System.** This module deals with the clients that connect, send requests and receive responses from replicas. Given the open-nature of this communication (replicas can serve an unbounded number of clients) we choose the Netty communication framework [2] for implementing client/server communication. The most important requirement of this module is that it should be able to accept and deal with a few thousands of connections efficiently. To do this, the Netty framework uses the `java.nio.Selector` class and a configurable thread pool.

- **Client Manager.** After receiving a request from a client, the replica verifies the authenticity of a request and stores it to be ordered by the replication protocol. For each connected client, this module stores the sequence number of the last request received from this client (to detect replay attacks), the last reply sent to the client (to deal with retransmissions), and maintains a queue containing the requests received but not yet delivered to the service being replicated. The requests to be ordered in a consensus are taken from these queues in a fair way.

- **Server Communication System.** While the replicas accept connections from an unlimited number of clients, as is supported by the client communication system described above, the server communication system implements a closed-group communication model used by the replicas to send messages between themselves. The implementation of this layer was made through "usual" Java sockets, using one thread to send and one thread to receive for each server. One of the key responsibilities of this module is to reestablish the channels between every two replicas after a failure and a recovery.

3.1.2 State machine replication

The state machine replication core was implemented using the simple interface provided by the communication system to access reliable and authenticated point-to-point links. More specifically, BFT-SMaRt uses six main modules to achieve state machine replication.

- **Proposer**: this simple module (which contains a single class) implements the role of a proposer, i.e., it defines how to propose a value in a *PROPOSE* message and what a replica should do when it is elected as a new leader.

- **Acceptor**: this module implements the core of the consensus algorithm: *PROPOSE*, *ACCEPT* and *WRITE* messages are processed and generated (in the case of the latter two) here.

- **Total Order Multicast (TOM)**: this module gets pending messages received by the client communication system and calls the proposer module to start a consensus instance. Additionally, a class of this module is responsible for delivering requests to the service replica and to create and destroy timers for the pending messages of each client.

- **Execution Manager**: this module is closely related to the TOM and is used to manage the execution of consensus instances. It stores information about consensus instances and their rounds as well as who was the leader replica on these rounds. Moreover, the execution manager is responsible to stop and re-start a consensus being executed.

- **Leader Change Manager**: Most of the complex code to deal with leader changes is in this module. Although the rules for validation and verification of executed and pending requests are notoriously hard to understand and implement, the code of this module is sequential (i.e., a set of nested loops) and is not in the protocol critical path. This means that this code does not suffer from concurrency problems and neither needs to be very efficient.

- **Reconfiguration Manager**: The reconfiguration protocol is implemented by this module. To avoid unnecessary modifications in other parts of the code-base, this module provides a consistent view of the group of replicas in the system and the number of tolerated faults.

3.2 Staged Message Processing

A key issue when implementing a high-throughput replication middleware is how to break the several tasks of the protocol in an architecture that is robust and efficient at the same time. In the case of BFT SMR there are two additional requirements: the system should deal with hundreds of clients and resist malicious behaviors from both replicas and clients.

Figure 3 presents the main architecture with the threads used for staged message processing [34] of the protocol implementation. In this architecture, all threads communicate through *bounded queues* and the figure shows which thread feeds and consumes data from which queues.

The client requests are received through a thread pool provided by the Netty communication framework. We have implemented a request processor that is instantiated by the framework and executed by different threads as the client load demands. The policy for thread allocation is at most one per client (to ensure FIFO communication between clients and replicas), and we can define the maximum number of threads allowed.

Once a client message is received, it is checked whether it is an ordered or unordered request. Unordered requests

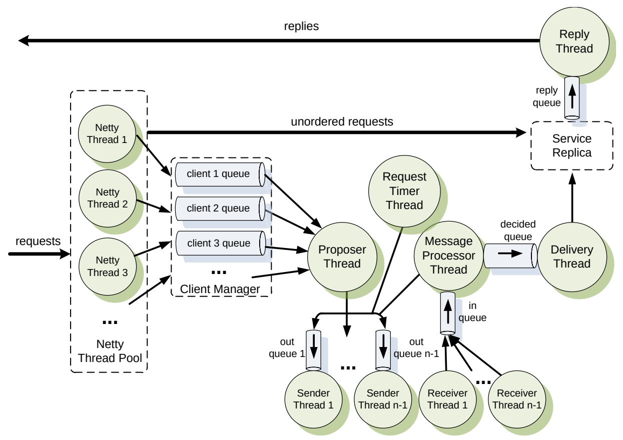


Figure 3: BFT-SMaRt replica staged message processing.

are directly delivered to the service implementation. Otherwise, they are delivered to the client manager, that verifies the request integrity and (if validated) adds them to the respective client's pending requests queue. Notice that since client' MACs and signatures (optionally supported) are verified by the *Netty threads*, multi-core and multi-processor machines would naturally exploit their power to achieve high throughput (verifying several client signatures in parallel).

The *proposer thread* waits for three conditions before starting a new instance of the consensus: (i) the replica is the leader for the next consensus; (ii) the previous instance is already finished; and (iii) at least one client (pending requests) queue has messages to be ordered. In a leader replica, the first condition will always be true, and it will propose a batch of new requests to be ordered as soon as a previous consensus is decided and there are pending messages from clients. Notice the proposal size will contain all pending requests (up to a maximum size, defined in the configuration file), so there is no waiting to fill a batch of certain size before proposing. In non-leader replicas, this thread is always sleeping waiting for condition (i).

Every message *m* to be sent by one replica to another is put on the *out queue* from which a *sender thread* will get *m*, serialize it, produce a MAC to be attached to the message and send it using TCP sockets. At the receiver replica, a *receiver thread* for this sender will read *m*, authenticate it (i.e., validate its MAC), deserialize it and put it on the *in queue*, where all messages received from other replicas are stored in order to be processed.

The *message processor thread* is responsible to process almost all messages of the state machine replication protocol. This thread gets one message to be processed and verifies if this message consensus is being executed or, in case there is no consensus currently being executed, it belongs to the next one to be started. Otherwise, either the message consensus was already finished and the message is discarded, or its consensus is yet to be executed (e.g., the replica is executing a late consensus) and the message is stored on the *out-of-context queue* to be processed when this future consensus is able to execute. As a side note, it is worth to mention that although the *PROPOSE* message contains the whole batch of messages to be ordered, the *WRITE* and *ACCEPT* messages only contain the cryptographic hash of this batch.

When a consensus is finished on a replica (i.e., the replica received more than $\frac{n+f}{2}$ *ACCEPT* messages for the same value), the decision is put on the *decided queue*. The *delivery thread* is responsible for getting decided values (a batch of requests proposed by the leader) from this queue, deserialize all messages from the batch, remove them from the corresponding client pending requests queues and mark this consensus as finalized. After that, the delivery thread invokes the *service replica* to make it execute and log the requests and generate the corresponding replies. When the batch is properly logged and the response is generated by the replica, the service replica adds the reply into the *reply queue*. The *reply thread* is responsible for sending the replies to the clients.

The *request timer thread* is periodically activated to verify if some request remained more than a pre-defined timeout on the pending requests queue. The first time this timer expires for some request, causes this request to be forwarded to the current known leader. The second time this timer expires for some request, the instance currently running of the consensus protocol is stopped and the synchronization phase is started (see §2.3.1). The rationale for these timers is the following: in normal network conditions, a timeout may be caused either by a client that did not send the request to the leader or by a leader that did not ordered the client request. Since typically there are many clients and few servers, we expect to have much more faults among clients, so we first assume there was a problem with the client and the leader is suspected only if the problem persists.

4 API and Programming Model

To implement a service based on BFT-SMaRt two main classes are used. The `ServiceReplica` is used at server side to instantiate a BFT-SMaRt replica while the `ServiceProxy` is used at client side for accessing the replicated service. The instantiation of `ServiceReplica` requires the provision of a numeric id (which is mapped to an IP and port through a configuration file) and implementations of an `Executable` – which defines the methods called when the service needs to process a request – and a `Recoverable` – which defines the state management. Usually these two interfaces are implemented by a single class (see bellow). At the client side, the `ServiceProxy` requires only the numeric id of the client. In the following we present a brief overview of the BFT-SMaRt API and refer the interested reader to [1] for more information.

Server-side. The abstract class `DefaultSingleRecoverable` implements the `Executable` and `Recoverable` interfaces considering a simple state transfer manager based on logging and checkpoints. To use this manager, a developer needs to extend the class implementing the following abstract methods:

```
public byte[] executeOrdered(byte[] cmd, MsgContext ctx);
public byte[] executeUnordered(byte[] cmd, MsgContext ctx);
public byte[] getSnapshot();
public void installSnapshot(byte[] state);
```

BFT-SMaRt invokes both the `executeOrdered` and `executeUnordered` methods upon delivering client commands to the application. The former is invoked when clients issue ordered commands, and the latter is invoked for unordered ones (typically read-only operations). These methods must implement the service code and return replies to be sent to the client. The `cmd` argument represents the serialized command issued by the client, and `ctx` contains command metadata (e.g., the id of the client, the consensus instance where it was ordered, the latency of the consensus, etc). Additionally, `ctx` also contains a timestamp and a set of nonces which are equal in all replicas. These values are necessary in applications that need to access a local clock or generate random values; they should use these values instead, in order to preserve the determinism property required by SMR [13].

Moreover, developers also need to implement `getSnapshot` and `installSnapshot` to create and install serialized snapshots of the application state, respectively. This serialization implemented in `getSnapshot` must be done in a deterministic manner: the snapshot created by a replica *r* representing state *S*, must be equal to the snapshot created by any other replica to represent *S*.

The `DefaultSingleRecoverable` class is usually employed by most BFT-SMaRt-based services. However, an application can use custom implementations of `Executable` and `Recoverable`. For instance, the API provides some specializations of `Executable` with methods to make the service execute one request at time (the default behavior, as described above) or a batch of requests at once (returning several replies).

The `Recoverable` interface can be used to implement custom state transfer protocols. This class provides a set of callback methods called by the BFT-SMaRt core:

```
public ApplicationState getState(int eid, boolean sendState);
public int setState(ApplicationState state);
public StateManager getStateManager();
```

Developers need to implement `getState` and `setState` to create and define the application state, respectively. Notice these methods require an implementation of `ApplicationState`, an abstract representation of the service state. Moreover, `getStateManager` returns the strategy used to manage state transfer, which can also be implemented by programmers. These features are used to implement the techniques described in [9].

By default, BFT-SMaRt replies directly to the clients that issued the commands after ordering and executing their requests. However, it is possible to override this procedure by providing a custom `Replier` to the `ServiceReplica`. This can be used (together with asynchronous invocations – see bellow) to implement replicated forwarders (e.g., firewalls, publish-subscribe brokers), where one client (a sender) sends the request to be processed and the "reply" is sent to another client (a receiver) [19].

Client-side. At client side, each instance of `ServiceProxy` represents a single BFT-SMaRt client with a distinct id. This class provides the following methods to issue commands to the server:

```
public byte[] invokeOrdered(byte[] request);
public byte[] invokeUnordered(byte[] request);
public void invokeAsynchronous(byte[] request,
      ReplyListener listener, int[] receivers, MsgType type);
```

For all methods, commands and replies must be serialized into a byte array. `invokeOrdered` and `invokeUnordered` are used to issue ordered and unordered commands, respectively. The `invokeAsynchronous` method can be used to issue both types of commands in a non-blocking manner, i.e., the service proxy will return without waiting for the replicas' replies. This enables programmers to create applications that can resume their execution while the library collects replies in background. To use this feature programmers will have to provide a callback defined by the `ReplyListener` interface to explicitly manage the reception of replies. We used this feature, for example, to implement the client part of a variant of the Byzantium transaction processing protocol [20].

Finally, the client can also modify the way a BFT-SMaRt client vote server replies through the provision of custom `Comparator` (used to compare server replies) and `Extractor` (used to extract a reply from a set of consistent replies) implementations. This feature is used, for instance, to support the confidentiality mechanisms employed in the DepSpace coordination service, where the servers reply cryptographic shares of a tuple, and the client need to verify if they are compatible and extract the reply through a combination of them [8].

5 Alternative Configurations

As already mentioned in previous sections, by default BFT-SMaRt tolerates non-malicious Byzantine faults, as most work on BFT replication (e.g., [13, 23]). However, the system can be tuned to tolerate only crashes or (intelligent) malicious behavior.

5.1 Crash Fault Tolerance

BFT-SMART supports a configuration parameter that, if activated, makes the system strictly crash fault-tolerant (CFT). When this feature is active, the system tolerates $cv.f < cv.n/2$ (i.e., only a majority of correct replicas is required), which implies changes in all required quorums of the protocols, and bypasses one *all-to-all* communication step during the consensus execution (the *WRITE* step of the consensus is not required). Other than that, the protocol is the same as in the BFT case, with MACs still enabled for message verification, bringing also tolerance to message corruption even in CFT setups.

5.2 Intrusion Tolerance

Making a BFT replication library tolerate intrusions requires one to deal with several concerns that are not addressed by most BFT protocols [7]. Here we discuss how BFT-SMART deals with some of these concerns.

Previous works showed that the use of public-key signatures on client requests makes it impossible for clients to forge MAC vectors and force leader changes (making the protocol much more resilient against malicious faults) [5, 16]. By default, BFT-SMART does not use public-key signatures[4] other than for establishing shared symmetric keys between replicas, however the system optionally supports the use of signed requests for avoiding this problem.

These same works also showed that a malicious leader can launch undetectable performance degradation attacks, making the throughput of the system as small as 10% of what would be achieved in fault-free executions. Currently, BFT-SMART does not provide defenses against such attacks. However, the system can be easily extended to support periodic leader changes to limit damage [16]. In fact, the codebase of a very early version of BFT-SMART was already used to implement a protocol resilient to this kind of attack [31].

Finally, the fact that we developed BFT-SMART in Java makes it easily deployable in different platforms[5] for avoiding single-mode failures, caused by accidental events (e.g., a bug or infrastructure problems) or malicious attacks exploiting common vulnerabilities. Such compromises on the running platforms can be mitigated by the deployment of replicas in different operating systems [18] or even cloud providers [33].

6 Evaluation

In this section we present results from BFT-SMART's performance evaluation. These experiments consist of (1) some micro-benchmarks designed to evaluate the library's raw throughput and latency, (2) the comparison of this performance with some competing systems and (3) an experiment designed to depict the performance's evolution of a small application implemented with BFT-SMART once the system is forced to withstand events like replicas faults, state transfers, and system reconfigurations.

6.1 Experimental Setup

Unless stated otherwise, all experiments ran with three (CFT) and four (BFT) replicas hosted in separated machines. The client processes were distributed uniformly across another four machines. Each client machine ran up to eight Java processes, which in turn executed up to fifty threads which implemented BFT-SMART clients (for a total of up to 1600 client processes).

Clients and replicas executed in the Java Runtime environment 1.7.0_21 on Ubuntu 10.04, hosted in Dell PowerEdge R410 servers. Each machine has two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading, i.e., supporting 16 hardware threads, and 32 GB of memory. All machines communicate through an isolated gigabit Ethernet network.

6.2 Micro-benchmarks

"Standard" benchmarks. We start by reporting the results we gathered from a set of micro-benchmarks that are commonly used to evaluate state machine replication systems, and focus on replica throughput and client latency. They consist of a simple client/service implemented over BFT-SMART that performs throughput calculations at the server side and latency measurements at the client side. Throughput results were obtained from the leader replica, and latency results from a selected client (always the same). Figure 4 presents the results.

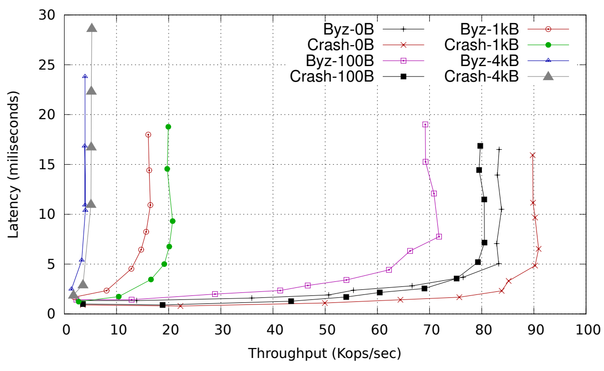
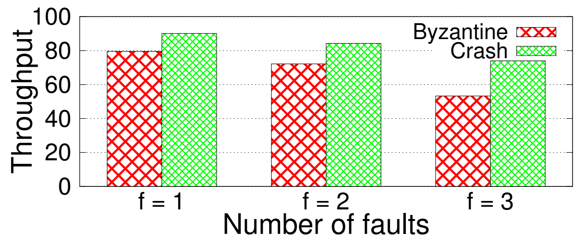


Figure 4: Latency vs. throughput configured for $f = 1$.

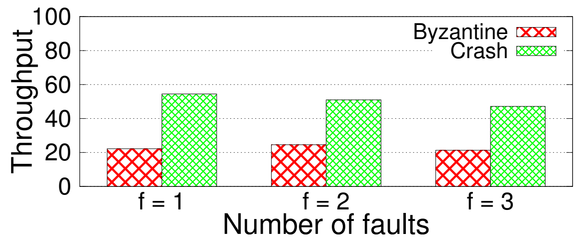
The figure illustrates BFT-SMART performance in terms of client latency against replica throughput for both BFT and CFT protocols. The standard deviation in all experiments was under 3%. For each protocol we exe-

[4]Client requests do not contain MAC vectors also, only point-to-point MACs as provided by the client communication system.

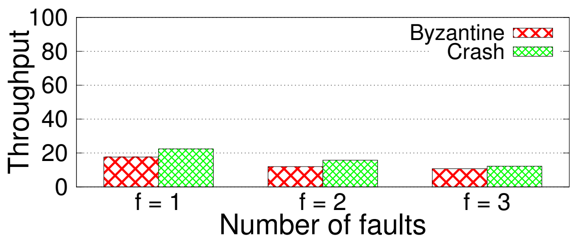
[5]Although we did not support N-versions of the system codebase, we believe supporting the deployment in several platforms is a good compromise solution.



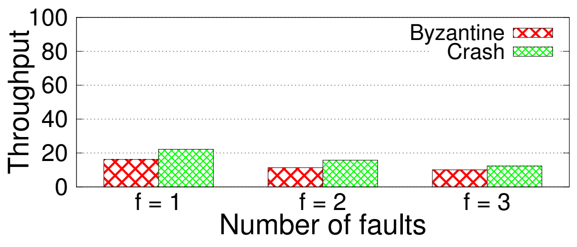
(a) 0/0



(b) 0/1024



(c) 1024/0



(d) 1024/1024

Figure 5: Peak sustained throughput (in Kops/sec) of BFT-SMART for CFT ($2f+1$ replicas) and BFT ($3f+1$ replicas) considering different workloads and number of tolerated faults.

cuted four experiments for different request/reply sizes: 0/0, 100/100, 1024/1024 and 4096/4096 bytes. Figure 4 shows that for each payload size, the CFT protocol consistently outperforms its BFT counterpart. This difference is due to the smaller number of messages exchanged in the CFT setup, which reflects in less work per client request for the replicas. Furthermore, as the payload size increases, BFT-SMART overall performance decreases. This is because (1) the overhead of requests/replies transmission between clients and replicas increases with message size, and (2) since Mod-SMaRt orders requests in batches, the larger is the payload, the bigger (in bytes) the batch becomes, thus increasing its transmission overhead among replicas.

We complement the previous results with Table 1, which shows how different payload's combinations affect throughput. This experiment was conducted under a saturated system running 1600 clients using only the BFT protocol. Our results indicate that increasing request's payload generates greater throughput degradation than reply's payload does. This can also be explained by the larger batch submitted to the consensus protocol, since request's payload influences its size, whereas reply's does not.

Requests \ *Replies*	0 bytes	100 bytes	1024 bytes
0 bytes	83337	75138	37320
100 bytes	78711	72879	36948
1024 bytes	16309	16284	15878

Table 1: Throughput for different requests and replies sizes for $f = 1$. Results are given in operations per second.

Fault-scalability. Our next experiment consider the impact of the size of the replica group on the peak sustained throughput of the system under different benchmarks. The results are reported in Figure 5.

The results show that, for all benchmarks, the performance of BFT-SMART degrades graciously as f increases, both for CFT and BFT setups. In principle, these results contradict the observation that protocols containing all-to-all communication patterns are less scalable as the number of faults tolerated [3]. This is not the case in BFT-SMART because (1) it exploits the many cores of the replicas (which our machines have plenty) to calculate MACs, (2) only the $n-1$ *PROPOSE* messages of the consensus protocol are big, the other $2n(n-1)$ messages are much smaller and contain only the hash of the proposed request batch, and (3) we avoid the use of IP multicast, which is know to cause problems with many senders (e.g., multicast storms) [10].

Finally, it is also interesting to see that, with relatively big requests (1024 bytes), the difference between BFT and CFT tends to be very small, independently on the number of tolerated faults. Moreover, the performance drops between tolerating 1 to 3 faults is also much smaller with big payloads (both requests and replies).

Mixed workloads. Figure 6 reports the results of our experiment considering a mix of read and write requests. In the context of this experiment, the difference between reads and writes is that the former issues small requests (almost-zero size) but gets replies with payload, whereas the latter issues requests with payload but gets replies with almost zero size. This experiment was also conducted under a saturated system running 1600 clients.

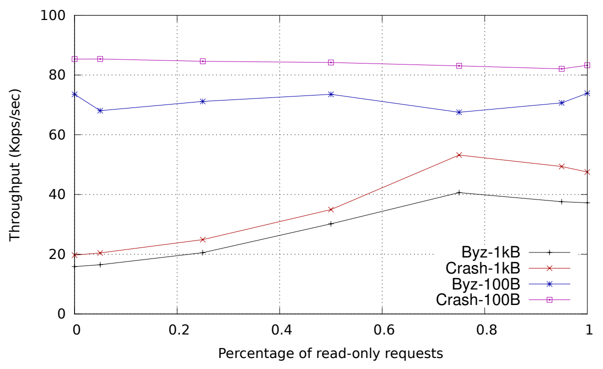


Figure 6: Throughput of a saturated system as the ratio of reads to writes increases for $n = 4$ (BFT) and $n = 3$ (CFT).

We performed the experiment both for the BFT and CFT setups of BFT-SMaRt, using requests and replies with payloads of 100 and 1024 bytes. Similarly to the previous experiments, the CFT protocol outperforms its BFT counterpart regardless of the ratio of read to write requests by around 5 to 15%. However, the observed behavior of the system regarding the throughput differs between the case of 100 bytes and 1024 payloads, with the former clearly benefiting from a larger read/write ratio.

This happens because 1024 bytes requests (a write operation) generate batches much larger than requests with only 100 bytes of payload. This in turn spawns a much greater communication overhead in the consensus protocol. Therefore, as we increase the read to write ratio for payloads of 1024 bytes, the consensus overhead decreases, which in turn improves performance. This happens with up to 75% reads, which has a better throughput than 95%- or 100%-read workloads. This happens for payloads of 1024 bytes because at this point sending the large replies of the read become the contention point of our system. Notice this behavior is much less significant with small payloads.

Signatures and Multi-core Awareness. Our next experiment considers the performance of the system when signatures are enabled, and used for ensuring resilience to malicious clients [16]. In this setup a client signs every request to the replicas that first verify its authenticity before ordering it. There are two fundamental service-throughput overheads involved in using 1024-bit RSA signatures. First, the messages are 112 bytes bigger than when SHA-1 MACs are used. Second, the replicas need to verify the signatures, which is a relatively costly computational operation.

Figure 7 shows the throughput of BFT-SMaRt with different number of threads being used for verifying signatures. As the results show, the architecture of BFT-SMaRt exploits the existence of multiple cores (or multiple hardware threads) to scale the throughput of the system. This happens because the signatures are verified by the Netty thread pool, which uses a number of threads proportional to the number of hardware threads in the machine (see Figure 3).

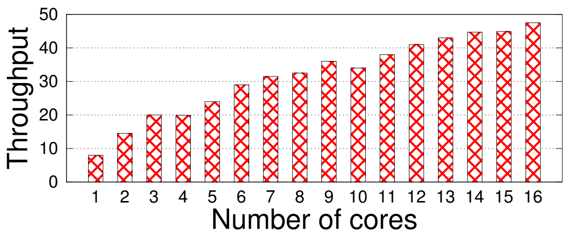


Figure 7: Throughput of BFT-SMaRt (in Kops/sec) using 1024-bit RSA signatures for 0/0 payload and $n = 4$ considering different number of hardware threads.

Comparison with others. We compared BFT-SMaRt against some representative SMR systems considering the 0/0 benchmark. More precisely, we compared BFT-SMaRt (both in BFT and CFT setups) with PBFT [13], UpRight [15] and JPaxos [28] (a modern CFT replication library). All systems were downloaded from the internet[6] in October 2013, installed and configured to mimic the setup used in their respective papers. In the case of UpRight, we used four machines as servers, three of them with a replica and an ordering server and the last one with only an ordering server. Table 2 shows the *peak sustained throughput* obtained for all these systems and the associated number of clients required to achieve this throughput in our environment.

System	*Throughput*	*Clients*	*Throughput 200*
BFT-SMaRt	83801	1000	66665
PBFT	78765	100	65603
UpRight	5160	600	3355
CFT-SMaRt	90909	600	83834
JPaxos	62847	800	45407

Table 2: Peak sustained throughput (and associated number of clients used for reaching this value) of different replication libraries for the 0/0 benchmark and $f = 1$. *Throughput 200* reports the throughput obtained by these system with 200 clients.

The results presented in Table 2 show that, in our environment, BFT-SMaRt achieves higher throughput than both PBFT and JPaxos. Even though PBFT reaches its peak throughput with only 10% of the amount of clients required with BFT-SMaRt, it did not displayed higher throughput with more than 100 clients. We hypothesize that this happens because PBFT is single-threaded, which makes it very efficient with few clients but limits its scalability. Anyway, this result is consistent with recent reports about PBFT performance (e.g., [17]).

[6]Projects home pages: `http://www.pmg.csail.mit.edu/bft/`, `https://code.google.com/p/upright/` and `https://github.com/JPaxos/JPaxos`.

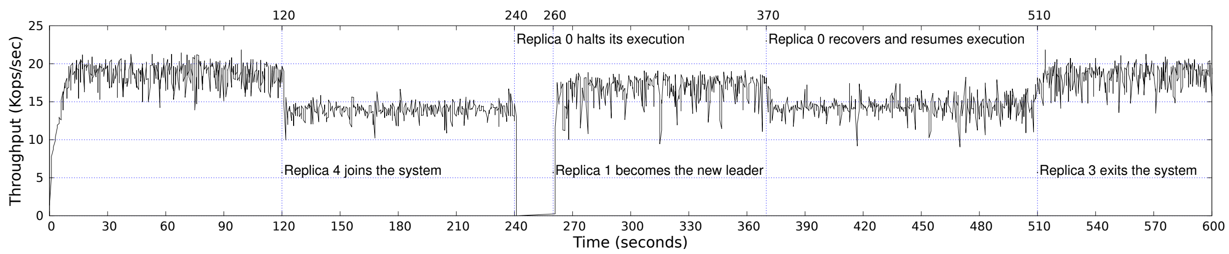


Figure 8: Throughput evolution across time and events, for $n = 4$ and $f = 1$.

JPaxos displayed a performance lower than what is reported in [28] (around 100 Kops/sec). Since we are using the same type of network, the only reason for that is that in the paper they use machines with 24 cores, while our servers support only 16 hardware threads.

As expected, the performance numbers obtained with UpRight were an order of magnitude lower than the others, which is consistent with the values presented in [15].

Following these results, we sought to get the performance values when the number of clients were the same for all libraries. The table also presents the throughput of the systems with 200 clients for each system.[7]

BFT-SMaRt displayed again the highest throughput under these conditions. However, notice that PBFT's performance decreased with twice the number of clients. This indicates that the system implementation suffers from some kind of trashing.

6.3 Faults, Reconfigurations, etc.

In this section we present an experiment designed to evaluate the behavior of an application implemented using BFT-SMaRt, and how it fares against replica's failures, recoveries, and reconfigurations. For this test we use the BFTMapList service, an in-memory table storing linked lists associated with each key. This is a simple (but non-trivial) data structure commonly used in practice (e.g., in social network applications).

BFTMapList implementation. BFTMapList is an implementation of the *Map* interface from the Java API which uses BFT-SMaRt to replicate its data in a set of replicas. It can be initialized at the client side providing transparency of the underlying replication mechanism. This is done by invoking BFT-SMaRt within its implementation. In BFTMapList, keys correspond to string objects and values correspond to a list of strings. We implemented the *put*, *remove*, *size* and *containsKey* methods of the aforementioned Java interface. These methods insert/delete a new String/List pair, retrieve the amount of values stored, and check if a given key was already inserted in the data structure. We also implemented an additional method called *putEntry* so that we could directly add new elements to the lists given their associated key.

[7]The choice of 200 clients was not arbitrary; this is the maximum number of clients supported by PBFT without crashing.

To evaluate this system, we created client threads that constantly insert new strings of 100 bytes to these lists, but periodically purge them to prevent the lists from growing too large and exhaust memory. Each thread corresponds to one BFT-SMaRt client.

Results. We sought to observe how BFTMapList performance would evolve upon several events within the system - ranging from replicas faults, leader changes, state transfers and system reconfigurations. For this experiment, BFT-SMaRt was configured with 4 replicas (with ids ranging from 0 to 3), to tolerate a single Byzantine fault. Our results are depicted in Figure 8, presenting throughput values collected from replica 1. We launched 30 clients issuing the *put*, *remove*, *size* and *putEntry* operations over the course of 10 minutes.

As the clients started their execution, the service's throughput increased until all clients were operational around second 10. At second 120 we inserted replica 4 into the service. As we did this, we observed a decrease in throughput. This can be explained by the fact that more replicas demand larger quorums in the consensus protocol and more messages to be processed in each replica. This reconfiguration spawns more message exchanges among replicas, which add congestion to the network and results in inferior performance.

At second 240, we crashed replica 0 (the current consensus' leader). As expected, the throughput dropped to zero during the 20 seconds (twice the timeout value configured in the system) that took the remaining replicas to trigger their timeouts and run Mod-SMaRt's synchronization phase. After this phase was finished, the system resumed execution. Since at this point there are less replicas executing, there are also less messages being exchanged in the system and the throughput was only slightly smaller than in the initial configuration.

At second 370, we restarted replica 0, which resumes normal operation after triggering the state transfer. Upon its recovery, the system goes back to the throughput exhibited before replica 0 had crashed.

At second 510, we removed replica 3, thus setting the quorum size to its original value, albeit with a different set of replicas. Since there is one less replica to handle messages from, we are able to observe the system's original throughput again by the end of the experiment.

7 Lessons Learned

More than five years of development and three generations of BFT-SMART gave us important insights about how to implement and maintain high-performance fault-tolerant protocols in Java. In this section we discuss some of the lessons learned on this effort.

7.1 Java as a BFT programming language

Despite the fact that the Java technology is used in most application servers and backend services deployed in enterprises, it is a common belief that a high-throughput implementation of a state machine replication protocol could not be possible in Java [15]. We consider that the use of a type-safe language with several nice features (large utility API, no direct memory access, security manager, etc.) that makes the implementation of secure software more feasible is one of the key aspects to be observed when designing a replication library. For this reason, and because of its portability, we choose Java to implement BFT-SMART. However, our experience shows that these nice features of the language when not used carefully can cripple the performance of a protocol implementation. As an example, we will discuss how object serialization can be a problem.

One of the key optimizations that made our implementation efficient was to avoid Java default serialization in the critical path of the protocol. This was done in two ways: (1) we defined the client-issued commands as byte arrays instead of generic objects, thus removed the serialization and deserialization of this field of the client request from all message transmissions; and (2) we avoid using standard object serialization on client requests, implementing instead a customized method (using data streams instead of object streams). This removed the serialization header from the messages and was specially important for client requests that are put in large quantities on batches to be decided by a consensus.[8]

7.2 How to test BFT systems?

Although distributed systems verification and debugging is a lively research area (e.g., [11,27,29]), there are still no tools mature enough to be used. Our approach for testing BFT-SMART is based on the use of JUnit, a popular unit testing tool. In our case we use it in the final automatic test of our build script to run test scripts that (1) setup replicas, (2) run some client accessing the replicated service under test and verify if the results are correct, and (3) kill the replicas in the end. This approach can be automated with the use of fault-injection frameworks and, in fact, one of such tools was recently used to test our system [27]. Notice that this is black-box testing: the only way to observe the system behavior is through the client. Similar approaches are being used in other distributed computing open-source projects like Apache Zookeeper.

Our JUnit-based test framework allows us to easily inject crash-faults on the replicas. However, testing the system against malicious behaviors is much more tricky. The first challenge is to identify the critical malicious behaviors that should be injected on up to f replicas. The second challenge is how to inject the code of the malicious behaviors on these replicas. The first challenge can only be addressed with careful analysis of the protocol being implemented. Disruptive code can be injected to the code using patches, aspect-oriented programming (through crosscutting concerns that can be activated on certain replicas) or simple commented code (which we are currently using). Our pragmatic test approach can be complemented with orthogonal methods such as the Netflix chaos monkey [6] to test the system on site.

It is worth to notice that most faulty behaviors can cause bugs that affect the liveness of the protocol, since basic invariants implemented in key parts of the code can ensure safety (e.g., a leader proposing different values to different replicas should cause a leader change, not a disagreement). This means that several recent efforts in verification of safety properties in distributed systems through model checking (e.g., [11]) does not solve the most difficult problem in our experience: liveness bugs.

Moreover, the fact that the system tolerates arbitrary faults makes it mask some non-deterministic bugs, or Heisenbugs, turning the whole test process even more difficult. For example, an older version of the BFT-SMART communication system losed some messages sporadically when under heavy load. The effect of this was that in certain rare conditions (e.g., when the bug happens in more than f replicas during the same protocol phase) there was a leader change, and the system blocks. We call these bugs *Byzenbugs*, since they are a specific kind of Heisenbugs that happen in BFT systems and that only manifest themselves if they occur in more than f replicas at once. Consequently, these bugs are orders of magnitude more difficult to discover (they are masked) and very complex to reproduce (they seldom happen).

7.3 Dealing with heavy loads

When testing BFT-SMART under heavy loads, we found several interesting behaviors that appear when a replication protocol is put under stress. The first one is that there are always f replicas that stay late in message processing. The reason is that only $n - f$ replicas are needed for the protocol to make progress and naturally f replicas will stay behind. A possible solution for this problem is to make the late replicas stay silent (and not load the faster replicas with late messages that will be discarded) and when they are needed (e.g., when one of the faster replicas fails) they synchronize themselves with the fast

[8] A serialized 0-byte operation request requires 134 bytes with Java default serialization and 22 bytes in our custom serialization.

replicas using the state transfer protocol (which runs more often that expected).

Another interesting observation is that, in a switched network under heavy load in which clients communicate with replicas using TCP, spontaneous total order (i.e., client requests reaching all replicas in the same order with high probability) almost never happens. This means that the synchronized communication pattern described in Figure 2 does not happen in practice. This same behavior is expected in wide-area networks. The main point here is that developers should not assume that client request queues on different replicas will be similar.

The third behavior that commonly happens in several distributed systems is that their throughput tends to drop after some time under heavy load. This behavior is called *trashing* and can be avoided through a careful selection of the data structures[9] used on the protocol implementation and bounding the queues used for threads communication.

7.4 Signatures vs. MAC vectors

Castro and Liskov most important performance optimization to make BFT practical was the use of MAC vectors instead of public-key signatures. They solved a technological limitation of that time. In 2007, when we started developing BFT-SMaRt we avoided signatures at all costs due to the fact that the machines we had access at that time created and verified signatures much slowly than the machines we used in the experiments described in §6: a 1024-bit RSA signature creation went from 15 ms to less than 1.7 ms while its verification went from 1 ms to less than 0.09 ms (a 10× improvement). This means that with the machines available today, the problem of avoiding public-key signatures is not so important as it was a decade ago, specially if signature verification can be parallelized (as in our architecture).

7.5 Maintenance & Robustness

Our experience with BFT-SMaRt showed us that implementing a robust BFT system is indeed hard. Several experienced developers that worked in our system mentioned that it was potentially the most complex codebase they had worked on, despite its reasonably modest size. The main observation of these developers was that, at first glance, many parts of the code appear to be unnecessary. The need for these parts was not obvious at first, but they were introduced to deal with bugs that appeared as BFT-SMaRt was used in more and more projects. This is a consequence of the well-known gap between protocol specifications and descriptions and the code required to implement them efficiently and robustly [14].

We believe BFT-SMaRt is arguably more robust and performant than other complete BFT systems (PBFT or UpRight) for a single reason: it is being maintained and constantly improved. Our view is that it is too hard to implement a BFT replication library at once. A more sound strategy is to keep building and improving the system, finding application scenarios and, in the case of academia, looking for opportunities for funding, publication and student projects as the software evolves.

Since the start of the project, BFT-SMaRt was used for implementing coordination services, key-value stores, a metadata service for a distributed file system, a transaction processing engine for replicated databases, an application-level firewall, a publish-subscribe middleware and a RADIUS-based authentication service.[10] The fact that most of these use cases were developed by different programmers provided a lot of feedback for evolving the system along the years.

8 Conclusions

This paper reported our effort in building the BFT-SMaRt state machine replication library. Our contribution with this work is to fill a gap in SMR/BFT literature describing how this kind of protocol can be implemented in a safe and performant way. Our experiments show that the current implementation already provides a very good throughput for both small- and medium-size messages.

The BFT-SMaRt system described here is available as open-source software in the project homepage [1] and, at the time of this writing, there are several groups around the world currently using or modifying our system for their research needs.

Acknowledgments. We warmly thank Marcel Santos, Paulo Sousa, Bruno Brito, André Nogueira, Vinicius Cogo, Ricardo Mendes, Miguel Garcia and all researchers and students who contributed and used BFT-SMaRt during the years. We also thank Fernando Ramos and Nikola Knezevic for their comments on previous versions of this paper. This work was partially supported by the EC through projects TClouds (FP7/2007-2013, ICT-257243) and MASSIF (FP7/2007-2013, ICT-257475) and also by FCT through the Multiannual Program (LaSIGE).

References

[1] BFT-SMaRt: High-performance byzantine fault-tolerant state machine replication. `http://code.google.com/p/bft-smart/`.

[2] The Netty project. `http://netty.io`.

[3] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. of the ACM SOSP'05*, 2005.

[9]For example, data structures that tend to grow with the number of requests being received should process searches in $\log n$ (e.g., using AVL trees) to avoid losing too much performance under heavy load.

[10]The list of projects and papers that used BFT-SMaRt can be found at `http://code.google.com/p/bft-smart/wiki/UsedInAndBy`.

[4] M. K. Aguilera, I. Keidar, D. Malkhi, J.-P. Martin, and A. Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 2010.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[6] C. Bennett and A. Tseitlin. Chaos monkey released in the wild. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`, 2012.

[7] A. Bessani. From Byzantine fault tolerance to intrusion tolerance (a position paper). In *Proc. of the 5th Workshop on Recent Advances in Intrusion-Tolerant Systems – WRAITS'11*, 2011.

[8] A. Bessani, E. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proc. of ACM EuroSys'08*, 2008.

[9] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC 2013*, 2013.

[10] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2), June 2009.

[11] P. Bokor, J. Kinder, M. Serafini, and N. Suri. Efficient model checking of fault-tolerant distributed protocols. In *Proc. of the 41st IEEE/IFIP Int'l Conf. on Dependable Systems and Networks – DSN'11*, 2011.

[12] C. Cachin. Yet another visit to Paxos. Technical Report RZ 3754, IBM Research Zurich, Nov. 2009.

[13] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Trans. Computer Systems*, 20(4):398–461, Nov. 2002.

[14] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proc. of the 26th ACM Symposium on Principles of Distributed Computing - PODC'07*, 2007.

[15] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *Proc. of the ACM SOSP'09*, 2009.

[16] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. of NSDI'09*, 2009.

[17] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC 2012*, 2012.

[18] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Analysis of operating systems diversity for intrusion tolerance. *Software - Practice and Experience*, 2013. to appear.

[19] M. Garcia, N. Neves, and A. Bessani. An intrusion-tolerant firewall design for protecting SIEM systems. In *Proc. of the Workshop on Systems Resilience – WSR'13*, 2013.

[20] R. Garcia, R. Rodrigues, and N. Preguica. Efficient middleware for Byzantine fault-tolerant database replication. In *Proc. of ACM EuroSys'11*, 2011.

[21] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proc. of ACM EuroSys'10*, 2010.

[22] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *Proc. of ACM EuroSys'12*, 2012.

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4), Dec. 2009.

[24] L. Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, May 1998.

[25] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, Mar. 2010.

[26] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *Proc. of ACM EuroSys'06*, 2006.

[27] R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo. Experiences with fault-injection in a Byzantine fault-tolerant protocol. In *Proc. of ACM/IFIP/USENIX Middleware'13*, 2013.

[28] N. Santos and A. Schiper. Achieving high-throughput State Machine Replication in multi-core systems. In *Proc. of the 33rd IEEE International Conference on Distributed Computing Systems – ICDCS'13*, 2013.

[29] J. Simsa, R. Bryant, and G. Gibson. dBug: Systematic evaluation of distributed systems. In *Proc. of the 5th Workshop on Systems Software Verification – SSV'10*, 2010.

[30] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proc. of the 9th European Dependable Computing Conference – EDCC'12*, 2012.

[31] G. Veronese, M. Correia, A. Bessani, and L. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. of the 28th IEEE Symposium on Reliable Distributed Systems – SRDS'09*, 2009.

[32] G. Veronese, M. Correia, A. Bessani, L. Lung, and P. Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1), Jan. 2013.

[33] M. Vukolić. The Byzantine empire in the intercloud. *ACM SIGACT News*, 41:105–111, Sept. 2010.

[34] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the ACM SOSP'01*, 2001.