# State Machine Replication for the Masses with BFT-SMaRt

Alysson Bessani, João Sousa
Faculdade de Ciências, Universidade de Lisboa, Portugal

Eduardo E. P. Alchieri
Universidade de Brasilia, Brazil

***Abstract*—The last fifteen years have seen an impressive amount of work on protocols for Byzantine fault-tolerant (BFT) state machine replication (SMR). However, there is still a need for 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 a flexible programming interface. When compared to other SMR libraries, BFT-SMaRt achieves better performance and is able to withstand a number of real-world faults that previous implementations cannot.**

## I. Introduction

The last fifteen years have seen an impressive amount of papers about Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) (e.g., [1]–[7], to cite just a few), but almost no practical use in real deployments. We believe this happens because there are no robust-enough implementations of BFT SMR available – only prototypes used to validate novel ideas from papers – which makes it difficult to deploy this kind of technique in practice. The general perception is that BFT protocols are too complex to implement, and since non-omission faults are rare, they can usually be dealt with simple techniques like checksums [8].

To the best of our knowledge, from all "BFT systems" proposed so far, only the early PBFT [2] and the more recent UpRight [4] implement a complete replication system. However, PBFT's architecture does not fully exploit modern hardware and UpRight exhibits a performance significantly lower than other systems. Moreover, both the PBFT and UpRight software packages 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 [9] – it seems that there is still no widely-accepted implementation that can be used to develop dependable services. As a result, every organization that requires such services needs to create its own implementation (e.g., [10]).

In this paper we describe BFT-SMaRt, a robust Java-based BFT SMR library which implements a protocol similar to PBFT. BFT-SMaRt targets both high-performance in fault-free executions and correctness if faulty replicas exhibit arbitrary behavior. Besides its robustness, BFT-SMaRt is the first BFT SMR library to support reconfigurations of the replica set [11] and to provide efficient and transparent support for durable services [12].

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 protocols for state transfer and reconfiguration. Additionally, the paper presents an evaluation of BFT-SMaRt, comparing it with previous systems and shedding light on some performance tradeoffs related to tolerance of crashes vs. Byzantine faults.

The paper is organized as follows: §II and §III describe the design of BFT-SMaRt and its implementation, respectively. §IV presents alternative configurations for BFT-SMaRt. §V describes an evaluation of our system. §VI highlights some lessons learned during the development and maintenance of the system. Finally, §VII presents our concluding remarks.

## II. 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 [13]. In 2009, such implementation was revamped to create a complete BFT SMR library, including features such as state transfer and reconfiguration. Nonetheless, only in 2011 we obtained funding to substantially improve the system in terms of functionality and robustness.

### A. Design Principles

*Tunable fault model:* By default, BFT-SMaRt tolerates *non-malicious Byzantine faults*, a realistic (albeit pessimistic) system model in which (1) messages can be delayed, dropped or even corrupted, and (2) processes can execute abnormally and indulge in any spurious action. All these behaviors have been observed in real systems and components (see [8] for an overview). Besides that, BFT-SMaRt provides both cryptographic signatures (for improved tolerance to *malicious Byzantine faults*) and a simplified SMR protocol similar to Paxos [9] (to tolerate only crashes and message corruptions[1]).

*Simplicity:* The emphasis on protocol correctness led us to avoid optimizations that could bring extra complexity both in terms of deployment, coding, or even new corner cases. For this reason, we avoid techniques that, although promising in terms of performance (e.g., speculation [6] and pipelining [5]) or resource efficiency (e.g., using trusted components [7] or IP multicast [2], [6]), would make our

[1]Unless stated otherwise, we focus on the BFT setup of the system. Crash fault tolerance is discussed later in §IV.

code more difficult to render correct (due to new corner cases) or deploy (due to lack of infrastructure support). This emphasis also made us choose Java instead of C/C++ as the implementation language. In §V 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 a *modular* SMR protocol that uses a well defined consensus primitive in its core [14]. 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 between them. While both protocols are equivalent at runtime, modular alternatives tend to be easier to implement and reason about, when compared to monolithic protocols. Besides the existence of modules for reliable point-to-point communication and 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.

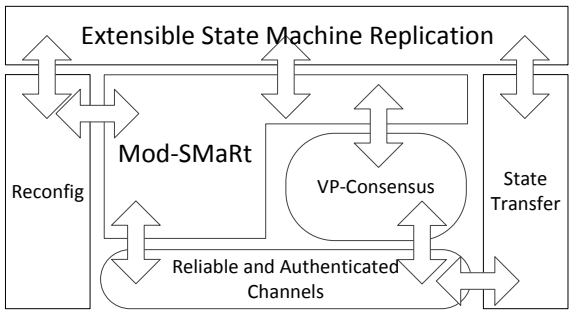



Figure 1. The modularity of BFT-SMART.

*Simple and Extensible Application Programming Interface:* Our library encapsulates all the complexity of BFT SMR inside a simple API that can be used by programmers to implement deterministic services. More precisely, if the service strictly follows the SMR programming model, clients can use a simple *invoke(command)* method to send commands to the replicas, that implement an *execute(command)* method to process the command. If the application requires specialized behaviors not supported by such a basic programming model, they can be implemented using 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, especially when signatures are enabled and more computing power is needed for their verification.

### B. System Model

BFT-SMART assumes the usual system model for BFT SMR [2], [5], [6]: $n \geq 3f+1$ replicas to tolerate $f$ Byzantine faults; an unbounded number of faulty-prone clients and eventual synchrony to ensure liveness. Moreover, since the system supports reconfiguration, it is possible to change $n$ and $f$ at runtime through *join* and *leave* operations (see §II-C3). Besides that, the system can also be configured to use only $n \geq 2f+1$ replicas to tolerate $f$ *crash faults* (see §IV-A). Independently of the configuration, the system requires reliable point-to-point links between processes for communication. These links are implemented using message authentication codes (MACs) over TCP/IP. The symmetric keys for the replica-replica channels are generated through Signed Diffie-Helman using a pair of RSA keys per replica. The keys for client-replica channels are generated based on the ids of the endpoints, without the need for clients to hold key pairs. Although this is in accordance with our assumption of non-malicious Byzantine faults, strong authentication of clients is still available if signed requests are enabled (see §IV-B).

### C. Core Protocols

In this section we give a brief overview of the protocols used by BFT-SMART and refer the interested readers to the papers describing them in detail [12], [14], [15].

*1) Total Order Multicast:* Total order multicast is achieved using Mod-SMaRt [14], a modular protocol which implements BFT SMR using an underlying consensus primitive. In particular, we use an extension of the leader-driven Byzantine consensus algorithm described in [15]. During normal execution, clients send their requests to all replicas and wait for their replies. Total order is achieved through a sequence of consensus instances, each of them deciding a batch of client requests. Each instance is comprised by three communication steps whose message pattern is illustrated in Figure 2. The first step requires the consensus' leader to send a *PROPOSE* message to other replicas. This is followed by two *all-to-all* communication steps consisting of *WRITE* and *ACCEPT* messages. Whereas *PROPOSE* messages contain the batch of requests to be decided, *WRITE* and *ACCEPT* messages only contain the cryptographic hash of such batch.

Figure 2 depicts Mod-SMaRt *normal phase* execution, which takes place in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, Mod-SMaRt may switch to its *synchronization phase*. During this phase, a new leader is elected for the consensus primitive and replicas are forced to "jump" to the same consensus instance. This jump might make some replicas trigger the state transfer protocol, described in the next section.

*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. Moreover, the possibility of

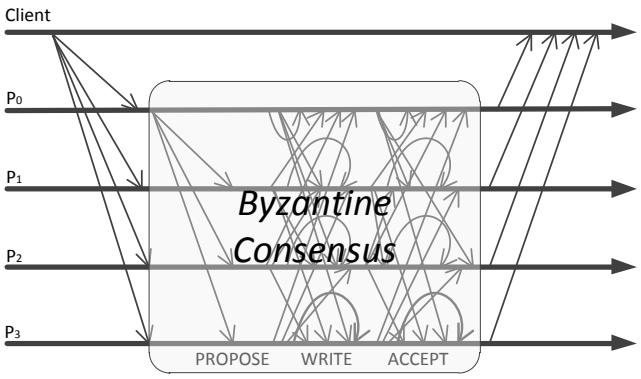



Figure 2. BFT-SMART normal phase message pattern.

correlated failures that can bring down more than $f$ replicas of the system at once requires the use of stable storage to recover the whole system. BFT-SMART implements the efficient durability techniques described in [12] to deal with the recovery of replicas or the whole system. The key ideas of such techniques are (1) to log batches of operations in a single disk while these operations are being executed by the service, (2) take snapshots at different points of the execution in different replicas to avoid stopping the system and (3) perform state transfer in a collaborative way, with each replica sending different parts of the state to the recovering replica. All these techniques are implemented in a well-defined layer between the replication protocol and the application, without influencing the consensus protocol.

*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 replicas to be added or removed from the system *on-the-fly*. Such process can only be initiated by system administrators running a *View Manager* client.

The reconfiguration protocol follows the ideas of [9], [11], but adapted to deal with Byzantine faults: the View Manager issues a special type of operation which is submitted to the Mod-SMaRt algorithm just like any other client operation. Through these operations, the View Manager notifies the system about the replica that it wants to add to (or remove from) the system. Since these operations are totally ordered (just like ordinary requests), all correct replicas will adopt the same view as the system's current view at any given point in the execution of client operations. Additionally, this operation must be signed with a special private key that guarantees the client is in fact a system administrator with privileges for reconfiguring the system.

After the View Manager operation is ordered, it is not delivered to the application like ordinary operations are. Instead, the request signature is verified to assess if it was produced using the view manager private key. If this signature is valid, the system current view is updated in accordance with the updates requested in the reconfigure operation. Following this, the replicas reply to the View Manager informing it about whether or not the view change succeeded. If it did, the View Manager sends a special message to the replica that is waiting to be added to (or removed from) the system, informing that it can either start or halt its execution. Lastly, if a replica is being added, it triggers the state transfer protocol to bring itself up to date.

All clients need to store the system's latest view in order to support reconfigurations. Therefore, replicas reject any client request issued in an old view, replying instead with data about the latest one. The clients then update themselves and retransmit theirs operation, now in the context of the latest view. Additionally, before accessing the system, a client must obtain the system's current view, which can be done with the use of a directory service.

## III. IMPLEMENTATION

BFT-SMART contains less than 13.5K lines of Java code distributed in little more than 90 files. This is significantly less than what was used in similar systems: PBFT [2] contains 20K lines of C code and UpRight [4] contains 22K lines of Java code. Even JPaxos [16], the most complete open-source CFT replication library we are aware of, contains more than 22K lines of Java code.

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. 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 of the protocol implementation. In this architecture, all threads communicate through *bounded queues*. The figure shows which thread feeds and consumes data from which queues.

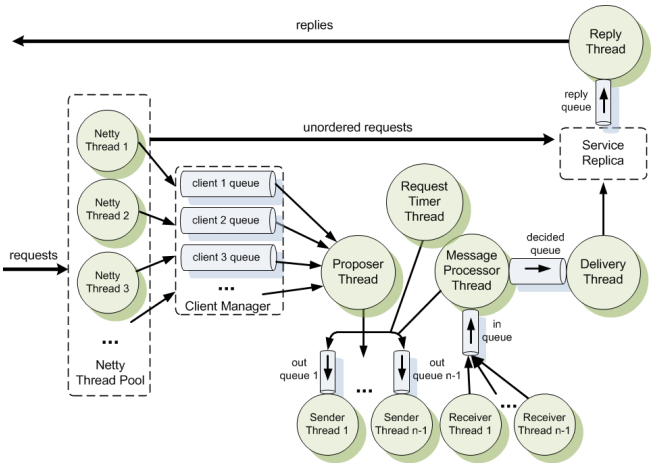



Figure 3. BFT-SMART replica staged message processing.

The client requests are received through a thread pool provided by the *Netty communication framework*. This design enables BFT-SMART to manage hundreds of client-to-replica connections efficiently. Once a client message is received, it is checked whether it is an ordered or unordered request. Unordered requests, which are usually employed for read-only commands [2], are directly delivered to the service implementation. Otherwise, they are delivered to the *client manager*, that verifies the request integrity and adds them to the respective client's queue. Notice that since clients' MACs and signatures (optionally supported) are verified by the *Netty threads*, multi-core and multi-processor machines will naturally exploit their power to achieve high throughput (verifying several client signatures in parallel).

The *proposer thread* is responsible for assembling a batch of requests and transmitting the *PROPOSE* message of the consensus protocol. BFT-SMART fills the batch with pending requests until: (a) its size either reaches a limit defined in a configuration file; or (b) it has no requests left to add. This thread is only active at the leader replica.

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 for processing messages from the BFT SMR protocol. This thread fetches messages from the *in queue* and processes them if they belong to the consensus currently being executed. If messages belong to a consensus ahead of the current one, they are processed later, when such consensus is triggered. Otherwise, the messages are discarded.

When a consensus is finished on a replica, the decided batch is put on the *decided queue*. The *delivery thread* is responsible for getting batches from this queue, deserialize all requests from the batch, remove them from the respective client queues and mark the current consensus as finalized. After that, the delivery thread invokes the *service replica* to execute the requests and generate the corresponding replies. After generating the reply, the *service replica* adds it into the *reply queue*. The *reply thread* fetches replies from this queue and sends them to the respective 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, it causes this request to be forwarded to the current known leader. The second time this timer expires for some request, the current instance of the consensus protocol is stopped and the synchronization phase is triggered (see §II-C1). 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 many 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 (see [14] for details).

## IV. Alternative Configurations

As mentioned in previous sections, by default BFT-SMART tolerates non-malicious Byzantine faults, as most works on BFT replication do (e.g., [5], [6]). However, the system can be configured to support two other fault models.

### A. 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 $f < n/2$ (simple minority), which implies changes in all required quorums of the protocols, and bypasses the *WRITE* step during the consensus execution. Other than that, the protocol is the same as in the BFT case.

### B. Malicious Byzantine Faults

Previous works showed that the use of public-key signatures on requests makes it impossible for clients to forge MAC vectors and force leader changes (making the protocol much more resilient against malicious faults) [1], [3]. By default, BFT-SMART does not use public-key signatures other than for establishing shared symmetric keys between replicas and during leader change. However the system optionally supports usage of signed requests to avoid this problem.

These same works also showed that a malicious leader can launch undetectable performance degradation attacks, making the throughput of the system drop dramatically. Currently, BFT-SMART does not provide defenses against such attacks. However, the system can be easily extended to support periodic leader changes to limit such damage [3].

Finally, the fact that we developed BFT-SMART in Java makes it easily deployable in different platforms.[2] This choice lets us avoid some single-mode failures caused by accidental events (e.g., a bug or infrastructure problems) or malicious attacks exploiting common vulnerabilities.

## V. 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) a performance comparison with some competing systems; and (3) an experiment showing the performance of a BFT-SMART-based service when withstanding faults and reconfigurations.

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

*Experimental Setup:* Unless stated otherwise, all experiments ran with three (CFT) and four (BFT) replicas hosted in separate machines. Up to 1600 client processes were distributed uniformly across another four machines.

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

*Micro-benchmarks:* We start by reporting the results of a set of micro-benchmarks commonly used to evaluate state machine replication systems. Such benchmarks consist of an "empty" service implemented with BFT-SMaRt to perform raw throughput calculations at the server side and latency measurements at the client side. Throughput measurements were gathered from the leader replica, while latency results from one of the clients (always the same).

Figure 4 presents results for both BFT and CFT setups of BFT-SMaRt considering different request/reply sizes: 0/0, 100/100, 1024/1024 and 4096/4096 bytes. In the figure it is possible to see that the CFT protocol consistently outperforms its BFT counterpart. This happens due to the smaller number of messages exchanged in the CFT setup, which results in less work per client request for the replicas. Furthermore, as expected, as the payload size increases, BFT-SMaRt overall performance decreases.

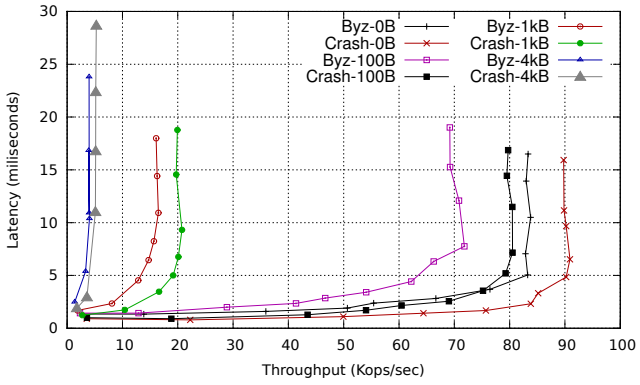
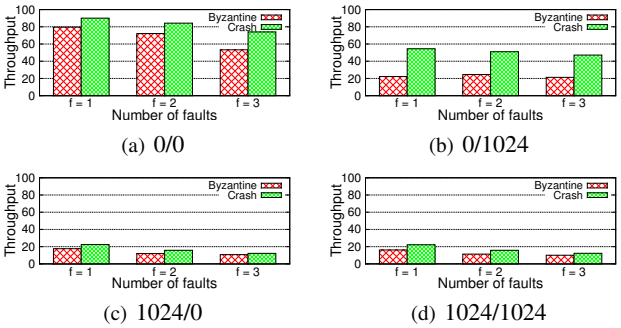



Figure 4. Latency vs. throughput configured for $f = 1$.

*Fault-scalability:* Our next experiment considers the impact of the number of replicas on the throughput of the system with different payloads. Figure 5 reports the results.

For all configurations, the results show that the performance of BFT-SMaRt degrades graciously as $f$ increases, both for CFT and BFT setups. This happens 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 contain batches of messages (the other $2n(n-1)$ messages exchanged during consensus only contain the hash of the batches); and (3) we avoid the use of IP multicast, which is know to cause problems with many senders (e.g., multicast storms) [17].

It is also interesting to see that, with relatively big requests (1024 bytes), the difference between BFT and CFT tends to be very small, regardless of the number of tolerated faults.




(a) 0/0 (b) 0/1024

(c) 1024/0 (d) 1024/1024

Figure 5. Throughput of BFT-SMaRt (Kops/s) for CFT ($n = 2f+1$) and BFT ($n = 3f+1$) for different workloads and $f = 1...3$.

*Signatures and Multi-core Awareness:* Our next experiment considers the performance of the system when client signatures are enabled. In this setup, the clients sign every request to the replicas that first verify its authenticity before ordering it. There are two fundamental service-throughput overheads associated with 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 6 shows the throughput of BFT-SMaRt with different number of hardware threads being used to verify signatures. As the results show, the architecture of BFT-SMaRt exploits the existence of multiple cores with hyper-threading. 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).

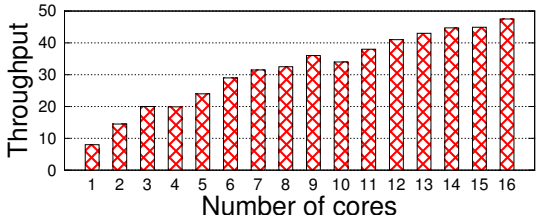



Figure 6. Throughput of BFT-SMaRt (in Kops/sec) using 1024-bit RSA signatures for 0/0 payload and $n = 4$.

*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 [2], UpRight [4] and JPaxos [16] (a modern multi-core CFT replication

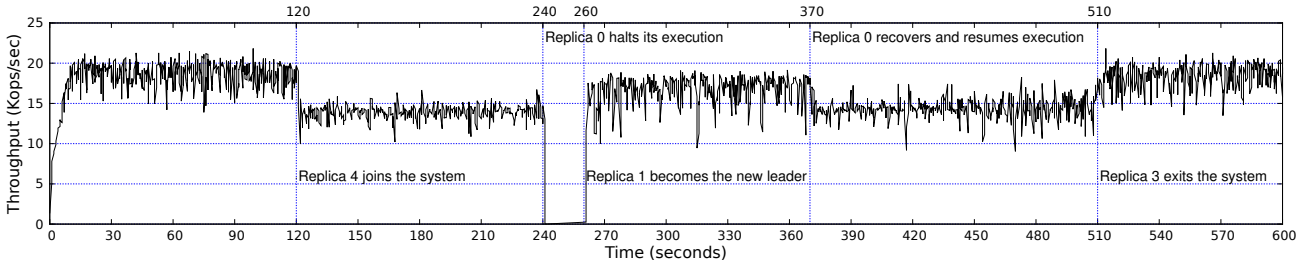



Figure 7. Throughput evolution across time and events, for $n = 4$ and $f = 1$.

library). All systems were downloaded from the internet[3] in October 2013, installed and configured to mimic the setup used in their respective papers. In the case of UpRight, we used three machines, each with a replica and an ordering server, plus another machine with just an ordering server. Table I shows the *peak sustained throughput* obtained for all these systems and the associated number of clients required to achieve this throughput in our environment.

The results presented in Table I show that BFT-SMaRt achieves a peak sustained throughput higher than both PBFT and JPaxos in our environment. Even though PBFT reaches its peak throughput with only 10% of clients required by BFT-SMaRt, it did not display 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. Nonetheless, this result is consistent with recent reports about PBFT performance (e.g., [8]). JPaxos displayed a performance lower than what is reported in [16] (around 100 Kops/sec). Since we are using the same type of network, the only reason for that is their use of machines with more hardware threads than ours (24 vs. 16). The performance numbers obtained with UpRight were an order of magnitude lower than the others, which is consistent with the values presented in [4].

The table also presents the throughput of the systems using the same number of clients[4]. BFT-SMaRt displayed again the highest throughput under these conditions.

| *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 I
SUSTAINED THROUGHPUT (AND 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 WITH 200 CLIENTS.

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

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

*Faults, Reconfigurations, etc.:* Our final experiment considers the behavior of an application implemented using BFT-SMaRt, and how it fares against replica's failures, recoveries, and reconfigurations. For this test we implemented a BFT-SMaRt service supporting an in-memory hashtable whose elements are linked-lists comprised by text strings (similar to the data structures used in social networks). The service was deployed in four replicas with ids from 0 to 3.

We observed the throughput of the service (measured at replica 1) evolve over several events within the system when a demanding workload is applied. We use 30 BFT-SMaRt clients that keep inserting, reading and deleting strings of 100 bytes in such lists over the course of 10 minutes. The result is presented in Figure 7.

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 system. With five replicas there is a throughput drop, since more replicas demand larger quorums in the consensus protocol and more messages are processed in each replica.

At second 240, we crashed replica 0 (the leader) and the throughput dropped to zero until the remaining replicas trigger Mod-SMaRt's synchronization phase (which takes 20 seconds). After the new leader takes over, the system resumes execution with a throughput only slightly smaller than in the initial configuration.

At second 370, we restarted replica 0, which resumes normal operation after executing a (very fast) state transfer. Upon its recovery, the system goes back to the throughput exhibited before replica 0 had crashed. Finally, at second 510, we return the system to four replicas through the removal of replica 3. Since there is one less replica to handle messages from, we are able to observe the system's original throughput again until the end of the experiment.

## VI. 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 knowledge we acquired during this period.

### A. Java as a BFT programming language

Even though Java technology is used in most application servers and backend services deployed in enterprises, it is common belief that a high-throughput implementation of a SMR protocol could not be possible in Java [4]. We consider that one of the key aspects to take into account when designing a replication library is the usage of a type-safe language. Additionally, such language must have several features that makes the implementation of secure software more feasible (e.g., large utility API, no direct memory access, security manager). For this reason, and because of its portability, we chose Java to implement BFT-SMaRt. However, our experience shows that these features, 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 removing the serialization and deserialization of this field from all message transmissions; and (2) we avoid using standard object serialization on client requests, implementing instead a customized method (using data streams rather than object streams). This removed the serialization header from the messages and was fundamental to reduce the size of batches decided by a consensus.[5]

### B. How to test BFT systems?

Although distributed systems tracing, debugging and verification is a lively research area (e.g., [18], [19]), there are still no tools mature enough to be used. Our approach for testing BFT-SMaRt is based on JUnit, a popular unit testing tool. In our case we use it in the final automated 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 [19]. 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 trickier. 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 code for 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 using patches, aspect-oriented programming (through crosscutting concerns that can be activated on certain replicas) or simple commented code (which we currently use). Our pragmatic test approach can be complemented with other methods such as the Netflix chaos monkey [20] to test the system on site.

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., [18]) do not solve liveness bugs, which is the most difficult problem in our experience.

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 lost 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) a leader change occured, but 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 very seldom happen).

### C. 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.

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 *thrashing* and can be avoided through a careful selection of the data structures[6] used on the protocol implementation and bounding the queues used for threads communication.

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

[6] 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.

### D. 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 the code required to implement them efficiently and robustly [10].

We believe BFT-SMART is arguably more robust and performs better 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 2007, BFT-SMART was used to implement prototypes of 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. 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.

## VII. 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 efficient 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 [21] and, at the time of this writing, there are several groups around the world using or modifying the system for their needs.

*Acknowledgments:* We warmly thank Paulo Veríssimo, Ricardo Fonseca, Pedro Costa, Fernando Ramos, Nikola Knezevic, Paulo Sousa, Marcel Santos, Bruno Brito, André Nogueira, Vinicius Cogo, Ricardo Mendes, Miguel Garcia and all students and researchers who contributed to and used BFT-SMART. This work was partially supported by the EC through projects TClouds (FP7-257243) and SEGRID (FP7-607109) and also by FCT through the LaSIGE Strategic Project (PEst-OE/EEI/UI0408/2014).

## References

[1] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[2] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[3] 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.

[4] 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.

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

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, Dec. 2009.

[7] G. Veronese, M. Correia, A. Bessani, L. Lung, and P. Verissimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, Jan. 2013.

[8] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems," in *Proc. of USENIX ATC 2012*, 2012.

[9] L. Lamport, "The part-time parliament," *ACM Transactions Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[10] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live - an engineering perspective (2006 invited talk)," in *Proc. of the PODC'07*, 2007.

[11] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, Mar. 2010.

[12] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. of USENIX ATC 2013*, 2013.

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

[14] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Proc. of EDCC'12*, 2012.

[15] C. Cachin, "Yet another visit to Paxos," IBM Research Zurich, Tech. Rep. RZ 3754, Nov. 2009.

[16] N. Santos and A. Schiper, "Achieving high-throughput State Machine Replication in multi-core systems," in *Proc. of the IEEE ICDCS'13*, 2013.

[17] K. Birman, G. Chockler, and R. van Renesse, "Toward a cloud computing research agenda," *SIGACT News*, vol. 40, no. 2, Jun. 2009.

[18] P. Bokor, J. Kinder, M. Serafini, and N. Suri, "Efficient model checking of fault-tolerant distributed protocols," in *Proc. of IEEE/IFIP DSN'11*, 2011.

[19] 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.

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

[21] "BFT-SMaRt project homepage," http://code.google.com/p/bft-smart/.