Byzantine Fault-Tolerant State Machine Replication with Twin Virtual Machines

Fernando Dettoni*, Lau Cheuk Lung*, Miguel Correia† and Aldelir Fernando Luiz‡
*Department of Informatics and Statistics
Federal University of Santa Catarina, Florianópolis, Brazil – Email: {fdettoni, lau.lung}@inf.ufsc.br
†INESC-ID, Instituto Superior Técnico
Technical University of Lisbon, Lisboa, Portugal – Email: miguel.p.correia@ist.utl.pt
‡Department of Automation and Systems
Federal University of Santa Catarina, Florianópolis, Brazil – Email: aldelir@das.ufsc.br

***Abstract*—The reliability and availability of distributed services can be ensured using replication. We present an architecture and an algorithm for Byzantine fault-tolerant state machine replication. We explore the benefits of virtualization to reliably detect and tolerate faulty replicas, allowing the transformation of Byzantine faults into omission faults. Our approach reduces the total number of physical replicas from $3f+1$ to $2f+1$. It is based on the concept of twin virtual machines, which involves having two virtual machines in each physical host, each one acting as failure detector of the other.**

***Index Terms*—Byzantine fault tolerance; intrusion tolerance; state machine replication; distributed systems; virtualization**

I. Introduction

Computing systems have been becoming more and more critical, so they have to operate correctly even in presence of faults. There are several classes of faults, ranging from accidental crash faults to arbitrary faults, often called Byzantine [1]. The latter is the class that is more generic so the one that should be considered in the design of critical systems. In other words, to ensure that these systems stay reliable and available it is necessary to develop Byzantine fault tolerance (BFT) mechanisms.

One of the most used fault tolerance approaches in this context is *state machine replication* (SMR) that consists essentially in replicating deterministic state machines in different hosts [2]. Several BFT SMR algorithms were developed in the past decade (e.g., [3], [4], [5], [6], [7]). PBFT [3] was the first that can be considered to be practical, therefore it is a kind of a baseline in the area, from which many later algorithms evolved [4], [5], [6], [7].

Virtualization can also be considered a Byzantine fault tolerance technique, in the sense that it introduces an isolation layer between virtual machines. Several approaches use virtualization to protect some components from others' failure or intrusion [8], [9], [10]. On the contrary to SMR and failure detectors, virtualization is widely adopted by industry; for instance, it is an essential technology in cloud computing services such as Amazon Web Services and Windows Azure.

This work was partially supported by the FCT through project RC-Clouds (PTDC/EIA-EIA/115211/2009) and contract PEst-OE/EEI/LA0021/2011 (INESC-ID) and CNPq Proc. 560258/2010-0.

PBFT and other BFT SMR algorithms have a considerable implementation cost because they require $n \geq 3f+1$ replicas to tolerate up to f faulty. To reduce this cost, some algorithms use a *trusted component* to constrain the behavior of faulty replicas, requiring only $n \geq 2f+1$ replicas [11], [6], [7]. Two recent approaches run only $f+1$ replicas and keep $2f$ replicas on standby, with lower consumption of CPU time, but requiring some delay to activate standby replicas in case a failure is suspected [12], [13].

This paper explores another point of the design space. We present *TwinBFT*, a new efficient BFT SMR algorithm based on virtualization. We reduce the number of physical machines from $n \geq 3f+1$ to $n \geq 2f+1$, to tolerate f faults. Furthermore, we reduce the number of communication steps in the normal case from 5 (as PBFT and related algorithms) to 3, without speculation [5]. To our knowledge, this is the first non-speculative algorithm with this number of steps. Speculation has the drawbacks of involving the participation of the client in the agreement and the ability to rollback operations [5], [7].

Our approach consists in using a set of *twin virtual machines*. Every virtual machine executes the same service and each pair of twins runs in one of $n \geq 2f+1$ physical hosts. Each pair of twins plays the traditional role of a replica in a SMR service. The main idea is to use each virtual machine as a *failure detector* of its twin: both twins must provide the same reply to every request, otherwise the twins suspect of each other, the physical node (a replica) is considered faulty and its messages are ignored by the rest. This way, Byzantine faults are transformed into omissions that are tolerated by the *state machine replication* algorithm. Host and virtual machine *crashes* are particular cases of omissions, which are tolerated the same way. Notice that we consider twin virtual machines for the sake of simplicity; the approach might be trivially generalized to any number of virtual machines per host.

We do not defend that TwinBFT is the best solution for all scenarios. It seems adequate for companies open to the use of virtualization, such as cloud computing service providers. It may also be appealing for companies reluctant to the use of trusted components or that cannot wait for the activation of standby replicas in the case of failure. In these cases,

an efficient BFT SMR algorithm with only $2f + 1$ physical hosts ($4f + 2$ virtual machines) is an interesting solution. Nevertheless, we do believe that companies today are very open to the use of virtualization, often reluctant to relying on trusted components, and sometimes unwilling to wait for replica activations.

The paper is organized as follows. Section II gives an overview of related work. Section III describes the system model and assumptions. A detailed explanation of the algorithm is given in Section IV. Section V presents the evaluation of algorithm and the Section VI concludes the paper.

II. Related Work

Several works on BFT SMR appeared in the last decade. PBFT is often considered to be the first practical algorithm in the area [3]. Although its performance seems adequate for many applications, its costs are considerable, requiring at least 4 replicas ($n = 3f+1$ with $f = 1$) and 5 communication steps in normal operation (worse in case there are faults). Several algorithms have evolved PBFT with two goals: reduce the number of replicas and improve performance.

a) Reducing the number of replicas: Yin *et al.* introduced an architecture separating services in two layers: one responsible for agreement, with $3f + 1$ replicas; another one for executing the requests, with only $2f + 1$ replicas [4].

Correia *et al.* presented the first solution to execute a BFT SMR with only $2f + 1$ replicas, using a trusted distributed component [11]. Later, another work presented the first algorithm requiring only $2f + 1$ replicas based on a trusted local component, using the abstraction of *attested append-only memory* [6]. Recently, Veronese *et al.* [7] proposed two algorithms based on a simple trusted component that just supplies unique message identifiers. The first one, MinBFT, reduced the number of necessary replicas to $2f + 1$ and the number of communication steps to 4. The second, a speculative version called MinZyzzyva, reduced the communication steps even further, to 3, keeping the number of replicas in $2f + 1$.

SMIT takes advantage of virtualization to reduce the number of replicas to $2f + 1$, leveraging the secure communication between replicas provided by the VMM / hypervisor [14]. That approach, however, requires all replicas to run on the same physical host, so it does not tolerate *crash* faults on the physical machine, unlike the approach in this paper.

Another work based on the idea of two replicas watching each other is presented in [15]. That work is based on the notion of *signal-on-fail*. The algorithm presented needs $4f+2$ physical machines and requires a synchronous (and trusted) communication channel between each pair of replicas, which is problematic to implement in practice.

b) Improving performance: Several works presented solutions to improve the performance of PBFT. Cowling *et al.* presented HQ, a quorum based protocol with very good performance when there is no concurrency to access data units [16]. Kotla *et al.* presented Zyzzyva, an algorithm able to reduce the number of communication steps in the absence of faults [5]. Instead of trying to reach an agreement before sending the reply to the client, the service replies speculatively. The service needs to execute the request again and reach an agreement if the replies received by the client differ from each other. This approach is efficient in executions that are free of failures, but requires the the ability to rollback operations, something that is not possible in many services.

c) Virtualization: Several works use virtualization to isolate software components. Two of the first use virtualization to protect an intrusion detector from intruders [9], [10], and a more recent one uses the same idea to protect a honeypot monitor [8]. Nevertheless, the hypervisor security is mandatory to obtain isolation so some works studied how to improve this security. Murray *et al.* proposed disaggregation of the virtualization system as a solution to reduce the size of the trusted computing base of the system [17]. NoHype goes further by removing the hypervisor of the way and executing the virtual machines natively on the hardware [18].

III. System Model

The architecture of the system is presented in Figure 1. The system is composed by a set of n physical hosts $H = \{h_1, h_2, \ldots, h_n\}$, where $n \geq 2f + 1$ and f is the maximum number of faulty hosts at any time. Each host contains a VMM (virtual machine monitor) or hypervisor with two virtual machines (VMs), called *twin virtual machines*, running one process each. Both processes $\{p_i, p'_i\}$ execute the same service, and communicate between each other to validate each message before sending it to other processes. We assume that virtualization provides isolation between the VMs and the VMM / hypervisor.

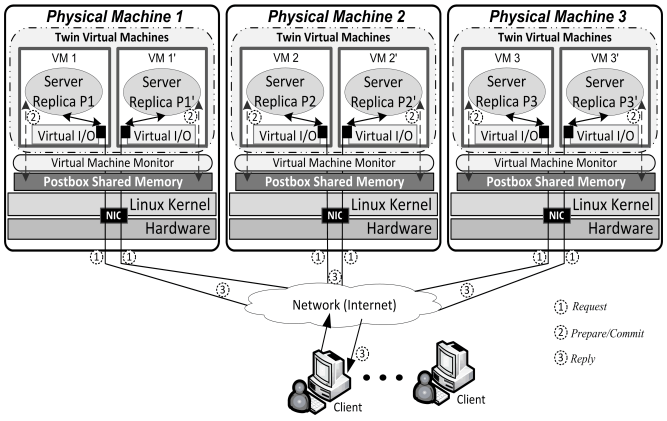


Fig. 1. The twin virtual machines architecture.

We assume that at most f VMs can fail arbitrary (or "be Byzantine"), but no more than one in the same physical host. We say that such process is *faulty*, otherwise we say it is *correct*. When a process is faulty, the validation mechanism transforms this fault into an omission of its host. Therefore, we also assume that up to f physical hosts can be *faulty*, but only by crashing (stopping to operate) or by omission (not sending some of the messages it should). These faults can be accidental or due to an arbitrary failure in one of it is processes/VMs.

In practice the assumptions about the limit of f faults and that no two processes are faulty in the same host can only be enforced optimistically by resorting to software diversity, i.e., to different implementations of the process at each replica [19], [20]. This diversity reduces the chance of more than one VM at the same physical host being compromised simultaneously.

No assumptions are made about the time needed to compute a request. The communication between different VMs inside the same host is made through a shared memory space, called *postbox*. The processes at different hosts communicate through the network, by message passing only. This network can fail to deliver, deliver out of order, delay, or duplicate messages. We make the same weak synchrony assumption about communication as PBFT for liveness [3].

Each host can assume two different roles: (1) *primary* host, which is responsible for defining the order for executing clients' requests; and (2) *backup* host, which executes the requests following the order proposed by the primary. Within a primary host, a process can assume two possible roles: (1) *leader*, which is responsible for assign the sequence number for client's requests; and (2) *follower*, which executes the requests following the order defined. All the processes within backup hosts are considered followers. The primary host h_i is the one for $i = v \bmod |S|$, where v is the current view (details later). The primary leader process within a server is, by definition, p_i.

We use cryptographic techniques to authenticate messages and ensure the authenticity of messages. Each pair of processes share among each other a secret key used to generate a vector of MACs (message authentication codes) [21] with a valid MAC for each process. We call these MACs signatures and say that a message with a vector of MACs is signed.

As mentioned in the introduction, we consider only the case of each physical machine having two VMs. This model, however, can easily be generalized to more VMs, following the condition of $nVM \geq 2fVM + 1$, where fVM is the maximum number of faulty virtual machines at the same host. In this case, a host will be considered faulty if a majority of VMs ($fVM + 1$) returns the same reply.

IV. TwinBFT

TwinBFT implements state machine replication in a set of processes and hosts. The replicas move through a succession of configurations called *views*. In each view, there is a primary/leader replica p_j that is responsible for defining the request order and forward it to all replicas. The state machine has to be deterministic and all replicas have to start in the same state [2]. In this section we explain TwinBFT in comparison to PBFT, following a common approach in the BFT literature.

A. Properties

Following the state machine replication approach, our algorithm has to ensure the following properties:

- *Total Order* (safety): a request is executed sequentially and in the same order on every replica;
- *Termination* (liveness): a request issued by a client is eventually executed, regardless of the existence of faults.

Our algorithm provides both safety and liveness, assuming that no more than $f = [\frac{n-1}{2}]$ hosts are faulty and there is at least one correct process p in each host. To ensure that all replicas will execute the requests in the same order, all the replicas follow the order defined by the leader and the leader can be assumed correct if the order proposed by the leader is signed by both processes at the primary host. A consensus algorithm is not necessary because the rest of the replicas monitor the behavior of the leader and change it if it misbehaves (details later). Our protocol ensures *safety* regardless of timing, but to ensure *liveness* we need the weak synchrony assumption mentioned in Section III.

B. The Algorithm

In this section, we will discuss the TwinBFT algorithm in detail. In Figure 2 we show a time diagram of the algorithm normal operation to help understanding it. The figure uses $f = 1$, so there are three hosts, each one with two VMs. The pairs of VMs communicate using the postbox, which is faster than the network by using a shared memory abstraction provided by the VMM.

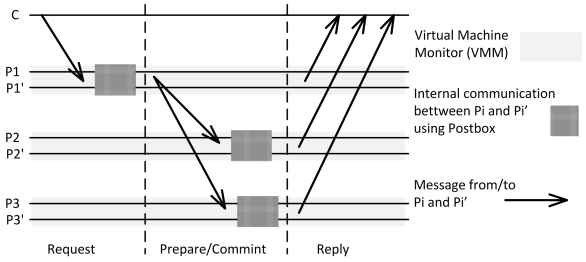


Fig. 2. Time diagram of TwinBFT in normal operation with $f = 1$.

The algorithm works basically as follows:

1) Client issues a request to both VMs in the primary host;
2) The primary's leader p_i defines a sequence number and posts an "ORDER" message on the postbox;
3) The primary's follower p'_i reads the message from the postbox, gets the sequence number and posts on postbox an "ORDER" message with the sequence number received;
4) Both VMs sign the "ORDER" read from their twin and send it to the backup replicas;
5) As soon as each VM inside a backup replica receive the message "ORDER", they execute the operation, and post a signed "REPLY" on the postbox;
6) When a VM reads a "REPLY", it compares with the one computed locally and if all fields matches, attaches its own signature to the message and sends it to client;
7) If the client receives at least $f+1$ correctly signed replies from distinct physical replicas, it accepts the result.

The part of the algorithm executed by the clients is shown in Algorithm 1. The client sends a request to the service (line 3) and waits until it receives at least $f + 1$ valid replies from

Algorithm 1 Client-side

```
1: procedure REQUEST                                   ▷ Issues new request
2:     Δ_c ← default timeout
3:     multi_send(⟨REQUEST, c, seq, op⟩_σc) ▷ Send request to both processes at the
   primary host
4:     repeat
5:         buffer ← buffer ∪ recv()
6:     until f + 1 matching replies ∃ buffer /* Timer in a separated
   thread */
7:     if Δ_c expired then
8:         multi_send(⟨REQUEST, c, seq, op⟩_σc)▷ Send the request to all the replicas
9:     end if
10: end procedure
```

Algorithm 2 Normal-case operation algorithm

```
/* Task 1: network */
1: loop
2:     msg ← receive()
3:     if received (REQUEST) then
4:         if is the primary leader then
5:             n ← n + 1
6:             postbox.append(⟨⟨ORDER, p_i, v, n, dm⟩_σpi, msg⟩)
7:         else if is the primary follower then
8:             buffer ← buffer ∪ msg
9:         else
10:            send msg to primary
11:            starts Δ_p
12:        end if
13:    else if received (ORDER) then
14:        stops Δ_p
15:        postbox.append(⟨REPLY, p_i, v, seq, c, res⟩ >_σpi)
16:    end if
17: end loop
    /* Task 2: postbox */
18: loop
19:    msg ← postbox.read()
20:    if received (ORDER) then
21:        if all parameters corresponds the ones locally computed then
22:            if is the primary follower then
23:                postbox.append(⟨⟨ORDER, p_i, v, m.n, dm⟩_σpi, msg⟩)
24:            end if
25:            multicast(⟨⟨⟨ORDER, p'_i, v, n, dm⟩_σp'i⟩σpi, msg⟩)
26:        end if
27:    else if received (REPLY) then
28:        if all parameters corresponds the ones locally computed then
29:            reply_to_client(⟨⟨REPLY, p'_i, v, seq, c, res⟩_σp'i⟩_σpi)
30:        end if
31:    end if
32: end loop
```

distinct replicas (lines 4-6). The request message has the form ⟨REQUEST, c, seq, op⟩$_{\sigma_c}$, where *c* is the client id, *seq* is a request id on the client, and *op* is the operation to be executed on the service. If the client does not receive $f + 1$ messages soon enough, it multicast the request to all replicas (line 8).

C. Normal Case Operation

The Algorithm 2, executed by the replicas, has two concurrent tasks. Task 1 is responsible for reading the messages received from the network. Task 2 is responsible for reading the messages from the postbox, posted by the twin. The state of each process is composed by the state of the service, a message buffer and the current view number. This state is shared among the tasks.

When the primary's leader process p_i receives a request from a client, it generates a new sequence number *n* and creates a message ⟨⟨ORDER, p_i, v, n, dm⟩$_{\sigma_{p_i}}$, m⟩, where *v* is the current view number, and *dm* is the signature of message *m* (lines 4-6). As soon as p'_i reads the p_i "ORDER" message from the postbox and has the "REQUEST" message in the message buffer, it gets the sequence number proposed by p_i, creates an "ORDER" message and posts it on the postbox (line 23). When each one reads an "ORDER" message from the postbox, it verifies if all parameters corresponds to the ones computed locally and, if yes, adds its own signature to the twin message and multicasts it to the backups (line 25).

A correct replica considers an "ORDER" message it received *valid* if:

- The message is correctly signed, i.e., if received from the network signed by both twin machines on the replica, and if received from the postbox signed by its twin process.
- The view in the message is the current view.
- The replica has not accepted another "ORDER" message with the same sequence number for a different request.
- The sequence number is between a low and high water marks h and H (in practice, if this verification is made when the primary's follower reads the "ORDER" message from postbox, a backup replica will never receive a message outside these water marks).

Upon a pair of twin processes receiving an "ORDER" message, each one verifies if the message is valid. If yes, it executes the operation and creates a message ⟨REPLY, p_i, v, seq, c, res⟩$_{\sigma_{p_i}}$, where *res* is the result of executing the operation, and posts it on the postbox (line 15). Once the other twin reads the "REPLY" from the postbox, it compares each parameter of the message with those it computed. If all parameters are identical, it signs the message generated by its twin and sends it to the client (line 29).

When the client receives a "REPLY" message, it accepts it as valid if the following conditions hold:

- Is is signed by the two processes of the sending host.
- The client has not yet received a valid reply to the same request from any of the processes on the physical host.

The client waits until it has received at least $f + 1$ valid messages from the replicas to accept the result. If it does not receive these messages soon enough, it multicast the "REQUEST" to all replicas (lines 7-9).

D. Garbage Collection

To prevent the system from running out of memory, TwinBFT has a mechanism to discard old messages stored on message buffers. To achieve this, the algorithm generates a checkpoint periodically, after some constant number of requests. To generate the checkpoint, each process generates a message ⟨CHECKPOINT, p_i, v, n, d⟩$_{\sigma_{p_i}}$, where *n* is the number of the last processed request and *d* is a signature of p_i current state, and posts it on the postbox.

Each twin machine reads the message from the postbox and as soon as it reaches the same checkpoint, it confirms if the state received is the same as the local state and, if yes, attaches its own signature in the "CHECKPOINT" message and multicasts it to all other replicas. When a process receives $f + 1$ "CHECKPOINT" messages properly signed and from distinct physical hosts h_i for the same view *v*, sequence number *n* and state *d*, it accepts this as the last valid checkpoint

Algorithm 3 View change algorithm

```
/* Task 1: network */
1: loop
2:     msg ← receive()
3:     if received (VIEW-CHANGE) then
4:         buffer ← buffer ∪ msg
5:         if buffer contains at least one VIEW-CHANGE with n = msg.n ∧ d = msg.d then
6:             send msg to primary
7:             if i = msg.v mod |S| then
8:                 postbox.append(⟨NEW-VIEW, p_i, msg.v, V, P⟩_σpi)
9:             end if
10:        end if
11:    else if received (NEW-VIEW) then
12:        buffer ← buffer ∪ msg
13:        for all req in msg.P do
14:            ensures req is processed and stored in its log.
15:        end for
16:    end if
17: end loop
/* Task 2: postbox */
1: loop
2:     msg ← postbox.read()
3:     if received f + 1 (VIEW-CHANGE) then
4:         if all parameters corresponds the ones locally computed then
5:             multi_send(⟨⟨VIEW-CHANGE, p'_i, v+1, n, C, P⟩_σp'i⟩_σpi)
6:         end if
7:     else if received (NEW-VIEW) then
8:         if all parameters corresponds the ones locally computed then
9:             multi_send(⟨⟨NEW-VIEW, p'_i, v+1, V, P⟩_σp'i⟩_σpi)
10:        end if
11:    end if
12: end loop
/* Task 3: timeout */
1: procedure TIMEOUT_EXPIRE                    ▷ When expiring timeout Δ_p
2:     postbox.append(⟨VIEW-CHANGE, p_i, v+1, n, C, P⟨_σpi)
3: end procedure
```

and removes from the message buffer all the messages with the sequence number lesser than *n*.

E. View Change Protocol

The main function of TwinBFT's view change protocol is to keep the service making progress even on the presence of a faulty primary. If the primary if faulty, the backup replicas may, for instance, not receive valid "ORDER" messages, so they must elect a new primary. Whenever a client does not receive enough valid replies to accept the result, it multicasts the request to all processes in the system. If a backup replica receives a request directly from the client, it verifies if it has already processed it. If yes, it just resends the reply sent previously; otherwise it forwards the request to both primary processes $\{p_i, p'_i\}$ and starts a local timer Δ_p (lines 10-11).

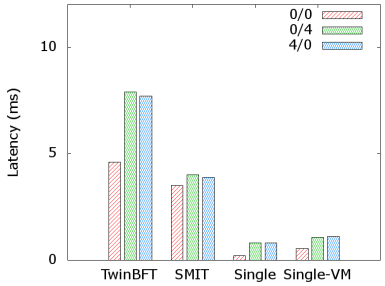
When receiving the corresponding "ORDER" message from the primary, the timer Δ_p is canceled (line 14) and the algorithm continues normally. If no "ORDER" message is received until the the timer expires, the process *p* starts the view change protocol of Algorithm 3, posting on the postbox a message $\langle$VIEW-CHANGE, p_i, v+1, n, C, P$\rangle_{\sigma_{p_i}}$, where *n* is the sequence number of the last valid checkpoint, *C* is a set composed by $f + 1$ "CHECKPOINT" messages asserting the last valid checkpoint, and *P* is a set with all the requests processed after the last checkpoint (line 2). If its twin agrees with the view change by verifying if the "VIEW-CHANGE" read from the postbox is equal to the one generated locally. If so, *p* attaches its own signature to the message received from the postbox and multicasts it to all processes (line 5).

Upon receiving $f + 1$ valid "VIEW-CHANGE" messages from a physical host, possibly their own, the two processes $\{p_i, p'_i\}$ verify if their host h_i is the new primary. If so, *p* acknowledges the view change by creating and posting on the postbox a message $\langle$NEW-VIEW, p, v+1, V, P$\rangle_{\sigma_p}$, where *V* is a set containing the "VIEW-CHANGE" messages sent by the servers for the view $v + 1$, and *P* is a set containing all the ORDER messages sent after the last valid checkpoint (lines 5-8. When *p* reads from the postbox a "NEW-VIEW", it verifies if it is according with the one computed locally and, if it is, signs and multicasts the message to all replicas (line 9).

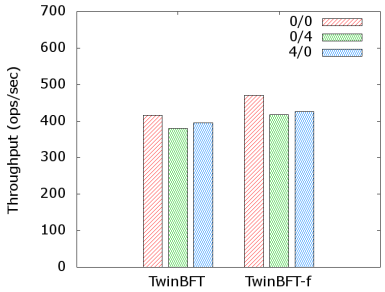
When any process *p* receives a "NEW-VIEW" message from the primary, it verifies if: (1) the message is properly signed, (2) it contains a set *V* with $f + 1$ valid "VIEW-CHANGE" messages. If the conditions are satisfied, it re-executes all the requests contained in *P* for the new view (lines 11-14).

V. Experimental Evaluation

For evaluation purposes, we developed a prototype for TwinBFT in Java 1.6, in which the communication channels were implemented through the Java NIO library, using TCP with MACs. We run our experiments on three servers Intel Core i7 3.8Ghz with Debian 7.0 "wheezy" (Kernel 3.2.0 x86-64) and the Xen Hypervisor 4.1.3. Each virtual machine was configured with 2GB of memory, 2 virtual CPUs, and SUN's JDK 1.6.0_29.



(a) Latency in normal operation.



(b) Throughput in normal operation and with faults.

Fig. 3. Latency and throughput in normal operation.

We evaluated the latency and throughput of TwinBFT, which are widely used metrics in the area as they give a simplified assessment of a system's efficiency [22]. The results were obtained using microbenchmarks with different load conditions.

Latency, the time needed to obtain the reply to a request, was obtained by having a single client sending one request at a time. Throughput was obtained by measuring how many requests the system can process by time unit. We evaluated the system using microbenchmarks due to their ability to measure the cost of running the system without the influence of the application/service. The service considered was a stateless service with null operations, with requests and reply sizes of 0KB and 4KB.

To evaluate the performance of algorithm, we execute the algorithm in normal operation and with view changes. We sent 10,000 requests from a single client, at three different loads: 0/0, 0/4 and 4/0. They represent, respectively, a null request and null reply, a null request and a 4KB reply, and a 4KB request and a null reply. All the times were measured by the client, by reading its local clock before issuing a request and after receiving a valid reply.

In Figure 3(a), we show the different latencies in each load. To obtain the latency, we sent requests individually and sequentially getting the latency from the average response time for all requests. The service provided by all the approaches is the same, a null operation returning the same message in the request to the reply. Our approach is compared with the SMIT algorithm [14], which also uses virtual machines and shared memory but does not tolerate crash faults. In the figure, the term *single* is used to mean an execution of a service without replication, hypervisor and virtual machines. *Single* is a centralized service without any kind of replication and the operations that are required to safely execute a replicated operation. Similarly, *single-VM* applies to an execution of a service inside a single virtual machine, also without replication. We can see that operations with large messages leads to a big increase in the response time. We can optimize this by letting the client choose one replica to send the complete reply while the other ones just send a digest to the client.

The throughput shown in Figure 3(b) was calculated based on the total time for the execution of 10,000 requests sent simultaneously to the service. The left of the figure shows the throughput in fault-free executions. The right ("TwinBFT-f") shows the throughput with 1% of requests affected by a faulty primary, resulting in a view change.

VI. Conclusions

The paper presents TwinBFT, a BFT SMR architecture and algorithm that leverages virtualization. TwinBFT requires only $2f + 1$ hosts, but does not need trusted components as the other algorithms that require the same number of hosts. Virtualization is currently a widely adopted technology, on the contrary of trusted components. TwinBFT has also a number of communication steps previously achieved only by speculative algorithms, which have constraints such as the need of rolling back operations.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[3] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 173–186.

[4] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 253–267, October 2003.

[5] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin, "Zyzzyva: speculative Byzantine fault tolerance," *Commun. ACM*, vol. 51, pp. 86–95, 2008.

[6] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, October 2007, pp. 189–204.

[7] G. S. Veronese, M. Correia, A. N. Bessani, L. C., and P. Verissimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[8] X. Jiang and X. Wang, "Out-of-the-box monitoring of VM-based high-interaction honeypots," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, Sep. 2007.

[9] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2003.

[10] M. Laureano, C. Maziero, and E. Jamhour, "Intrusion detection in virtual machine environments," in *Proceedings of the 30th Euromicro Conference*, 2004, pp. 520–525.

[11] M. Correia, N. F. Neves, and P. Verissimo, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, 2004, pp. 174–183.

[12] T. Distler, I. Popov, W. Schröder-Preikschat, H. P. Reiser, and R. Kapitza, "SPARE: Replicas on hold," in *Proceedings of the 18th Network and Distributed System Security Symposium*, Feb. 2011, pp. 407–420.

[13] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*, 2011, pp. 123–138.

[14] V. Stumm, L. C. Lung, M. Correia, J. da Silva Fraga, and J. Lau, "Intrusion tolerant services through virtualization: A shared memory approach," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 768 –774.

[15] D. Mpoeleng, P. Ezhilchelvan, and N. Speirs, "From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits," in *Proceedings of the IEEE/IFIP 33rd International Conference on Dependable Systems and Networks*, Jun. 2003, pp. 227–236.

[16] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006, pp. 177–190.

[17] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008, pp. 151–160.

[18] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 401–412.

[19] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, Jun. 2011, pp. 383 –394.

[20] I. Gashi, P. T. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, 2007.

[21] G. Tsudik, "Message authentication with one-way hash functions," *SIGCOMM Comput. Commun. Rev.*, vol. 22, no. 5, pp. 29–38, Oct. 1992.

[22] R. K. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.