How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems[*]

Miguel Correia Nuno Ferreira Neves Paulo Veríssimo
Faculdade de Ciências da Universidade de Lisboa
Bloco C6, Piso 3, Campo Grande, 1749-016 Lisboa - Portugal
{mpc,nuno,pjv}@di.fc.ul.pt

Abstract

The application of dependability concepts and techniques to the design of secure distributed systems is raising a considerable amount of interest in both communities under the designation of intrusion tolerance*. However, practical intrusion-tolerant replicated systems based on the state machine approach (SMA) can handle at most f Byzantine components out of a total of $n = 3f + 1$, which is the maximum resilience in asynchronous systems.*

This paper extends the normal asynchronous system with a special distributed oracle called TTCB. Using this extended system we manage to implement an intrusion-tolerant service based on the SMA with only $2f + 1$ replicas. Albeit a few other papers in the literature present intrusion-tolerant services with this approach, this is the first time the number of replicas is reduced from $3f + 1$ to $2f + 1$. Another interesting characteristic of the described service is a low time complexity.

1. Introduction

The application of dependability concepts and approaches to the design of secure distributed systems is raising a considerable amount of interest in both communities under the designation of *intrusion tolerance* [24]. The idea is that security concepts like vulnerability, attack and intrusion are contained in the dependability notion of fault, therefore it is possible to build secure systems based, to some extent, on dependability mechanisms. This idea has been used to design several protocols and systems in recent years [2, 3, 5, 11, 14, 16, 17, 19].

The *state machine approach* provides a general solution for the implementation of distributed fault-tolerant services [21]. The idea is to implement a service using a set of server replicas in such a way that the overall service can continue to behave as specified even if a number of servers is faulty. If the service is designed to tolerate arbitrary faults, which include attacks and intrusions, then the service can be said to be *intrusion-tolerant*, or Byzantine-resilient, since these faults are often called Byzantine[1].

This paper presents a solution for the implementation of *state machine replication services (SMR)* in practical distributed systems. The word *practical* is used in this context to signify open distributed systems with networks that provide weak quality of service guarantees, like the Internet, Ethernet LANs and other common network technologies. This kind of systems is often modelled using the *asynchronous model*, which makes no assumptions about processing times, communication delays or clock drift rates. The asynchronous model is extensively used mainly because it is hard to identify realistic bounds for these delays in practical systems. Moreover, for intrusion-tolerant systems, there is an additional motivation: protocols that make timing assumptions frequently have subtle vulnerabilities, which can be exploited in order to cause their failure [3]. We are aware of three asynchronous intrusion-tolerant SMR services in the literature: Rampart [19], BFT [3] and FS-NewTOP [17].

The *resilience* of a protocol can be defined as the maximum number of faults in the presence of which the protocol still behaves according to its specification. Notwithstanding the advantages of the asynchronous model discussed above, the optimal resilience for an SMR service based on this model is $\lfloor \frac{n-1}{3} \rfloor$, since the problem essentially boils down to atomic multicast [19, 3], which is equivalent to consensus [10]. A proof of the maximum resilience

* This work was partially supported by the FCT through project POSI/1999/CHS/33996 (DEFEATS), project POSI/CHS/39815/2001 (COPE) and the Large-Scale Informatic Systems Laboratory (LASIGE).

1 Throughout the paper we also use the expression *malicious faults* to emphasize that the cause of the fault is an intelligent attacker that has the purpose of violating some property of the system.

for asynchronous Byzantine consensus can be found in [1]. This means that the service needs $n > 3f$ replicas to tolerate f faults: four replicas to tolerate one fault, seven to tolerate two faults, etc. Each additional fault the system has to tolerate has a significative cost since it requires three additional machines. Moreover, the whole approach is based on the assumption that replicas fail independently, but this is true only if they do not have common vulnerabilities [3]. This involves using different replicas, i.e., different codes running in distinct operating systems. To summarize, each additional replica has two costs: (1) the cost of its hardware and software; and (2) the cost of its design, since it has to be different from the other replicas. Notice that the number of faults that can be tolerated can be improved either by detecting and removing faulty replicas [21], or by proactively recovering the state of the replicas [3]. However, in a window of time between detection and removal or between recoveries, the resilience remains $\lfloor \frac{n-1}{3} \rfloor$.

This paper presents a solution that reduces the cost of intrusion-tolerant SMR services by decreasing the number of replicas required to tolerate a number of faults/intrusions. More precisely, the presented SMR service has a resilience of $\lfloor \frac{n-1}{2} \rfloor$, i.e., it requires only a majority of correct replicas ($n > 2f$ servers to tolerate f faults). This means a reduction from 25% to 33% on the number of machines to tolerate the same number of faults: three replicas to tolerate one fault, five to tolerate two faults, seven to tolerate three faults, etc. Detection and removal, or proactive recovery of replicas, can also be used to improve the maximum number of faulty replicas.

How is it possible to improve the resilience from $f = \lfloor \frac{n-1}{3} \rfloor$ to $f = \lfloor \frac{n-1}{2} \rfloor$? The solution has something in common with the approach several protocols in the literature use to circumvent the Fischer, Lynch and Paterson (FLP) impossibility result [9]. FLP says that no deterministic protocol can solve the problem of consensus in an asynchronous system if a single process can crash. One of the most common approaches to circumvent this result is to extend the asynchronous system with some kind of *oracle*, like an unreliable failure detector [14, 12] or an ordering oracle [18]. These oracles allow the protocols to circumvent FLP because they encompass some degree of synchrony, e.g., enough synchrony to detect when a process crashed. The solution in this paper also relies on an oracle, but this particular oracle provides two advantages, instead of a single one: circumventing FLP and increasing the resilience.

In the past few years, we have been exploring a type of oracle called *wormholes* [22], to deal with the uncertainty (or lack of coverage) of assumptions such as time bounds [23] or intruder resistance [8, 5]. This paper extends the asynchronous system with a wormhole oracle called *Trusted Timely Computing Base (TTCB)*, already introduced elsewhere [8]. This oracle provides a novel ordering service that allow us to implement an atomic multicast protocol with a resilience of $\lfloor \frac{n-1}{2} \rfloor$. This service is the main building block of our SMR solution.

The paper provides the following main contributions:

- it presents an SMR service implemented mostly on a Byzantine asynchronous systems, but that uses the services provided by a oracle with stronger properties;
- the SMR service has a resilience of $\lfloor \frac{n-1}{2} \rfloor$ instead of the optimal resilience in asynchronous systems of $\lfloor \frac{n-1}{3} \rfloor$;
- the SMR service circumvents the FLP impossibility result without any synchrony assumptions on the asynchronous part of the system; all synchrony necessary to circumvent FLP is in the wormhole oracle;
- the service arguably exhibits good performance since it has a low time complexity.

2. System Model and the TTCB

The system is essentially composed by a set of hosts interconnected by a network, called payload network. This environment is asynchronous, i.e., there are no assumptions about processing delays or message delivery delays. The hosts have clocks but there are no assumptions, either about local clock drift rates, or about the reliability of the readings they provide.

The asynchronous environment is extended with a TTCB wormhole, a distributed component with local parts in some of the hosts (local TTCBs) and its own communication channel (TTCB control channel). The architecture of the system is depicted in Figure 1. Besides being distributed, the TTCB has three important features:

- it is assumed to be secure, i.e., resistant to any possible attacks; it can only fail by crashing;
- it is real-time, capable of executing certain operations with a bounded delay;
- it provides a limited set of services, which cannot be possibly affected by malicious faults, since the TTCB is secure.

The TTCB provides a very simple and limited set of services, so that the security of its implementation can be verified. This paper uses only two of these services. The first is the Local Authentication Service, which establishes a trusted path between the server and its local TTCB, i.e., a channel that guarantees the integrity of their communication [8]. The second is the Trusted Multicast Ordering service (TMO), which is the core of our solution and will be described in Section 3.

In relation to the real-time property mentioned above, it is important to make clear that the single consequence of

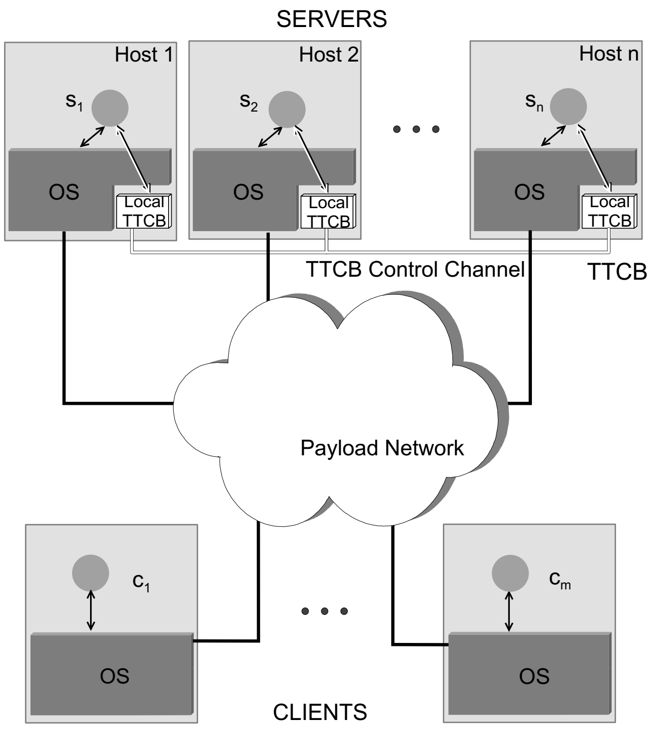

Figure 1. Architecture of the system.

this property for this paper is that the Trusted Multicast Ordering service is not bound by the FLP impossibility result. Otherwise there is no need for the TTCB to be synchronous in the context of this paper: the TMO can be implemented in a non-real-time wormhole if another solution is used to circumvent FLP, e.g., randomization or failure detectors.

The approach presented in the paper makes sense only if it is possible to implement the TTCB. There are several possible solutions, which were presented in another paper [8]. Moreover, an implementation based on COTS components is currently available for free noncommercial use[2]. Let us describe this implementation briefly for the reader to have an idea on how it works.

The local TTCBs have to be secure and real-time. The current TTCB implementation relies on an real-time engineering of Linux called RTAI [4] and is protected by hardening the kernel, since its code is executed inside the kernel. Another solution to protect the local TTCB would be to execute it inside a hardware module inserted in the computer, like a PC/104 board. In relation to the control channel, the current TTCB implementation relies on a Fast-Ethernet network, which is completely independent of the payload network (each host has two network adapters). The control-channel can be assumed to be secure for an inside premises system. Wide-area solutions could be based on virtual private networks over ISDN or Frame Relay. The real-time behavior is ensured by RTAI and by an admission control mechanisms that forces the control channel traffic to be limited and the communication delay bounded. This is a very brief idea and the reader is referred to [8] for a longer discussion on all these issues.

The SMR service is executed by a set of *servers* $S = \{s_1, s_2, ...s_n\}$. The service can be invoked by a set of *clients* $C = \{c_1, c_2, ...c_m\}$. The servers and clients are connected by a fully connected network, although their communication can be delayed arbitrarily, e.g., in consequence of an attack. Every host with a server needs a local TTCB, but not the hosts with clients (see figure). We use the word *processes* to denote both servers and clients. Each server s_i is uniquely identified by eid_i, a number obtained by calling the TTCB Local Authentication Service [8].

A process is *correct* if it follows the protocol it is supposed to execute. We assume that any number of clients can fail, but the number of servers that can fail is limited to $f = \lfloor \frac{n-1}{2} \rfloor$. The failures can be Byzantine or arbitrary, meaning that the processes can simply stop, omit messages, send incorrect messages, send several messages with the same identifier, etc. Faulty processes can pursue their goal of breaking the properties of the protocol alone or in collusion with other corrupt processes. A process is also considered to be faulty if one of the secret keys discussed below is disclosed, or if it is not able to communicate with the local TTCB, (e.g., due to a local denial of service attack).

The communication among clients and servers is done exclusively through the payload network. The communication among servers is also, to most extent, done through the payload network. We assume that each client-server pair $\{c_i, s_j\}$ and each pair of servers $\{s_i, s_j\}$ is connected by a *reliable channel* with two properties: if the sender and the recipient of a message are both correct then (1) the message is eventually received and (2) the message is not modified in the channel. In practice, these properties have to be obtained with retransmissions and using cryptography. Message authentication codes (MACs) are cryptographic checksums that serve our purpose, and only use symmetric cryptography [15]. The processes have to share symmetric keys in order to use MACs. In the paper we assume these keys are distributed before the protocol is executed. In practice, this can be solved using key distribution protocols available in the literature [15]. This issue is out of the scope of this paper.

Wrapping up, the system is essentially "asynchronous Byzantine": there are no bounds on the processing and communication delays; and the processes can fail arbitrarily. This system is extended with the TTCB wormhole, which is synchronous and secure, therefore it provides some "well-behaved" services that the processes can use to perform some steps of their protocols.

2 Available at http://www.navigators.di.fc.ul.pt/software/ttcb/

3. Trusted Multicast Ordering Service

The SMR service proposed in the paper uses a new TTCB service called *Trusted Multicast Ordering service (TMO)*. This service is implemented inside the TTCB, so its execution cannot be affected by malicious faults.

The TMO service was designed with the purpose of assisting the execution of an intrusion-tolerant atomic multicast (or total-order multicast) protocol. The service does *not* implement the atomic multicast protocol, but simply assigns an order number to the messages. The messages, however, are sent through the payload network, not through the TTCB. This is important since the TTCB has limited processing and communication capacities. Let us introduce briefly how an atomic multicast based on the TMO service can be implemented (the full protocol is introduced later in Section 4.2.1). When a process p wants to send a message to a set of recipients, it makes two operations: (1) it gives the TMO a cryptographic hash of the message and (2) it multicasts the message through the payload network reliable channels. Then, when another process q receives the message, it also gives the TMO a hash of the message it received. When a certain number of processes gave the hash of the message, the TMO service assigns an order number to the message and gives that number to the processes. The processes deliver the messages in that order. Figure 2 illustrates the procedure.

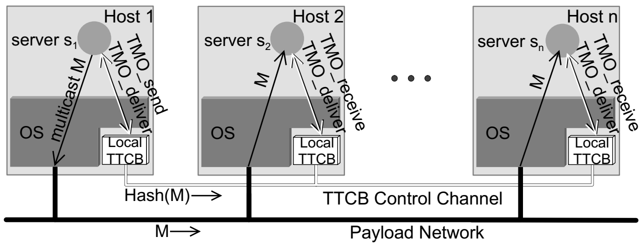

Figure 2. Atomic multicast using the TTCB TMO service.

The cryptographic hash mentioned above has to be obtained using a *hash function* h defined by the following properties [15]: *HF1 Compression:* h maps an input x of arbitrary finite length, to an output $h(x)$ of fixed length. *HF2 One way:* for all pre-specified outputs, it is computationally infeasible to find an input that hashes to that output. *HF3 Weak collision resistance:* it is computationally infeasible to find any second input that has the same output as a specified input[3]. *HF4 Strong collision resistance:* it is computationally infeasible to find two different inputs that hash to the same output.

The interface of the TMO service contains three functions: TTCB_TMO_send, TTCB_TMO_receive and TTCB_TMO_decide:

error, tag $\leftarrow$ TTCB_TMO_send(eid, elist, threshold, msg_id, msg_hash)

error, tag $\leftarrow$ TTCB_TMO_receive(eid, elist, threshold, msg_id, msg_hash, sender_eid)

error, order_n, hash, prop_mask $\leftarrow$ TTCB_TMO_decide(tag)

A process is said to start *an execution of the TMO service*, or simply to start *a TMO*, when it calls TTCB_TMO_send. The parameters of this function have the following meanings. The first, *eid*, is the identifier of the sender before the TTCB (see Section 2). *elist* is an array with the identifiers of all processes involved in the set of atomic multicast executions to be ordered. *threshold* is the number of processes in *elist* that must give the TTCB the correct hash of the message (*msg_hash*) for an order number to be assigned to the message. This parameter will be further discussed in Section 4.2.1. *msg_id* is a message number that has to be unique for the sender. *msg_hash* is a cryptographic hash of the message[4]. The function returns a *tag*, which identifies the TMO execution when the process later calls TTCB_TMO_decide, and an error code.

When a process receives a message it has to call TTCB_TMO_receive. The parameters are the same as for TTCB_TMO_send, except for the eid of the sender, *sender_eid*. How does the TTCB knows that a call to TTCB_TMO_receive corresponds to a certain TMO, which was started by a call to TTCB_TMO_send? It knows by looking at a set of parameters that together uniquely identify a TMO service execution: *(elist, threshold, msg_id, sender_eid)*. This last sentence has an important implication: if an attacker attempts to break the behavior of the TMO by calling TTCB_TMO_receive with any of these parameters modified, the TTCB will simply consider it to be a call to a different TMO, so the attack will be ineffective.

TTCB_TMO_receive returns a *tag* that is used by TTCB_TMO_decide to identify the TMO. When TTCB_TMO_receive is called and the local TTCB has no data about the TMO, a TMO_UNKNOWN error is returned. If there is data about the TMO but *msg_hash* is different from the hash provided by the sender, a

3 A guessing attack is expected to break the property HF3 in 2^m hashing operations, where m is the number of bits of the hash. A birthday attack can be expected to break property HF4 in $2^{m/2}$ hashing operations. In a practical setting, a hashing function with 128 bits like MD5, or 160 bits like SHA-1, can be considered secure enough for our protocol. Nevertheless, we consider HF2, HF3 and HF4 to be assumptions.

4 Later we use a value $\perp$ outside the range of valid hashes. A call to TTCB_TMO_send returns an error if *msg_hash*=$\perp$.

WRONG_HASH error is returned and the the call does not count for the threshold. If there is a TMO_UNKNOWN error, no *tag* is returned; on the contrary, if there is a WRONG_HASH error, the *tag* is returned.

To get the result of the TMO – the order number of the message – a process calls TTCB_TMO_decide. If *threshold* processes did not propose a hash equal to the hash proposed by the sender yet, a THRESHOLD_NOT_REACHED error is returned. If there is no error, the function returns the order number *order_n*, the hash of the message *hash* and a mask with one bit per process, indicating the processes that proposed the correct hash, *prop_mask*. For each TMO execution, the order number returned to all processes must be the same, since the TTCB is assumed secure and reliable.

The purpose of the TMO service is to assign consecutive numbers (1, 2, 3, ...) to a set of TMO executions. At this stage the reader might ask: does the TTCB orders *all* TMO executions with a single sequence of numbers? Or can there be several sets of TMO executions being ordered simultaneously by the TTCB? The answer is related to the purpose of the TMO service: to assist the execution of an atomic multicast protocol; there can be several atomic multicast channels in the system, therefore the TTCB has also to order several sets of TMO executions simultaneously. So, what TMO executions does the TTCB order? The TTCB orders independently each set of TMO executions that belong to the same sequence, defined as follows:

Two TMO executions, identified respectively by (elist$_i$, threshold$_i$, msg_id$_i$, sender_eid$_i$) and (elist$_j$, threshold$_j$, msg_id$_j$, sender_eid$_j$), are said to belong to the same sequence of TMO executions *iff elist$_i$ = elist$_j$.*

TMO Service Implementation A brief discussion of the implementation we envisage for the TMO service can give a sense of the semantics of the service. The protocol that implements the service is executed by all local TTCBs, which communicate using the TTCB control channel. The protocol can be simple because the TTCB is real-time, local TTCBs can only fail by crashing (they are secure) and they have synchronized clocks. The protocol is implemented on the top of the (crash-tolerant) reliable broadcast protocol presented in [7].

The protocol is based on a fixed coordinator. When a process calls TTCB_TMO_send or TTCB_TMO_receive in a local TTCB, the information about the call is broadcasted to all local TTCBs. When the coordinator gets information about *threshold* calls with the correct hash for a TMO execution, it assigns the next order number to the TMO, defines the mask *prop_mask* with the processes that proposed the correct hash, and broadcasts this information to all local TTCBs. Then, when a process calls TTCB_TMO_decide the order number is returned. If the coordinator crashes, another local TTCB takes over in a consistent manner, since it is aware of the broadcasts made by the coordinator.

4. State Machine Replication

A *state machine* is characterized by a set of *state variables*, which define the state of the machine, and a set of *commands* that modify the state variables [21]. Commands have to be atomic in the sense that they cannot interfere with other commands. The *state machine approach* consists of replicating a state machine in n servers $s_i \in S$. The set of servers S implements the *service*. We assume that no more than $f = \lfloor \frac{n-1}{2} \rfloor$ servers fail. All servers follow the same history of states if four properties are satisfied:

- *SM1 Initial state.* All servers start in the same state.
- *SM2 Agreement.* All servers execute the same commands.
- *SM3 Total order.* All servers execute the commands in the same order.
- *SM4 Determinism.* The same command executed in the same initial state generates the same final state.

The first property states that each state variable has the same initial value in all servers, something that is usually simple to guarantee. The second and third properties demand that the servers agree in the commands to execute and in their order. This can be guaranteed sending the commands to the servers using an atomic multicast protocol. The fourth property is about the semantics of the commands at the application level, so in this paper we simply make the assumption that the commands are deterministic.

The system works essentially the following way: (1) a client sends a command to one of the servers; (2) the server sends the command to all servers using an atomic multicast protocol; (3) each server executes the command and sends a reply to the client; (4) the client waits for $f + 1$ identical replies from different servers; the result in these replies is the result of the issued command. This is a very simplified description of the process, so let us first delve into the details of the clients, and later we describe the protocol executed by the servers.

4.1. Clients

A client c_i issues a command *cmd* to the service by sending a REQUEST message to one of the servers, s_j. The message is sent through the payload network, since the only communication that is performed inside the TTCB is the one related to the execution of the TMO service. The format of the message is:

$$\langle \text{REQUEST, addr, num, cmd, vec} \rangle$$

where REQUEST is the type of the message, *addr* is the address of the client (e.g., the IP address and the port), *num* is the request number, *cmd* is the command to be executed (including its parameters) and *vec* a vector of MACs (see discussion below). The request number has to be unique, since the SMR service discards requests from the same client with the same number. A solution to generate these numbers is to use a counter incremented for each sent message.

If the client and the server are correct, the REQUEST message is eventually received by s_j, due to the properties of the reliable channels (Section 2). Then, if the server is correct it atomically multicasts the message to all servers in S, all correct servers execute the command and send a reply to the client. The format of the reply message is:

$$\langle \text{REPLY, addr, num, res} \rangle$$

where REPLY is the type of the message, *addr* is the address of the server, *num* is the request number, and *res* is the result of the executed command.

This scheme, albeit simple, is vulnerable to some attacks. A server s_j can be malicious and forward the message only to a subset of S, or discard it altogether. To solve this problem, if c_i does not receive $f+1$ replies from different servers after T_{resend} units of time read in its local clock, it assumes that s_j did not forward the request, so it multicasts the message to another f servers. If this happens, it sends the message to a total of $f+1$ servers, therefore at least one must be correct, and the request will be atomically multicasted.

Ideally, T_{resend} should be greater than the maximum round trip delay between any client and a server. However, the payload system is assumed to be asynchronous, so there are no bounds on communication delays, and it is not possible to define an "ideal" value for T_{resend}. Therefore, the value of T_{resend} involves a tradeoff: if too high, the client can take long to have the command executed; if too low, the client can resend the command without necessity. The value should be selected taking this tradeoff into account. If the command is resent without need, the duplicates are discarded using a mechanism discussed in the next section.

A malicious server might attempt a second attack: to modify the message before multicasting it to the other servers. To tolerate this attack, the request message takes a vector of MACs *vec*. This vector takes a MAC per server, each obtained with the key shared between the client and that server. Therefore, each server can test the integrity of the message by checking if its MAC is valid, and discard the message otherwise[5].

In general, there will be restrictions on the commands that each client can execute. For instance, if the commands are queries on a database, probably not all the clients are allowed to query all registers in the same way. This involves implementing some kind of access control. There are several schemes available in the literature and this issue is orthogonal to the problem we are addressing in the paper, so we do not propose any particular scheme.

5 A malicious client might build a vector of MACs with a combination of valid and invalid MACs. This attack would be ineffective: if enough correct servers received the message with the correct MAC the command would be executed, otherwise it would be discarded.

4.2. Servers

The protocol executed by the servers is a thin layer on the top of an atomic multicast protocol. A server calls *atomic_mcast(*M_{REQ}*)* to atomically multicast a request M_{REQ} to all servers, and the atomic multicast protocol layer calls *atomic_dlv(*M_{REQ}*)* to deliver M_{REQ} to a server. This protocol is in Algorithm 1.

Algorithm 1 SMR protocol (for server s_i).

```
1: When a request M_REQ = ⟨REQUEST, addr, num, cmd, vec⟩
   is received from a client: if there is no message M_REQ', with
   M_REQ'.addr = M_REQ.addr and M_REQ'.num = M_REQ.num,
   for which atomic_dlv(M_REQ') has been previously called,
   then call atomic_mcast(M_REQ); otherwise discard the request.
2: When atomic_dlv(M_REQ) is called: if there is no message
   M_REQ', with M_REQ'.addr = M_REQ.addr and M_REQ'.num
   = M_REQ.num, for which atomic_dlv(M_REQ') has been previously called, then execute cmd and send a message ⟨REPLY,
   addr, num, res⟩ with the result of the command to the client.
```

The objective of checking in both steps if *atomic_dlv(*M_{REQ}*')*, with M_{REQ}*'.addr* = M_{REQ}*.addr* and M_{REQ}*'.num* = M_{REQ}*.num*, has been previously called, is to guarantee that a request from a client is executed only once. Recall that a client, even if correct, can resend a request (Section 4.1). This condition is implemented using a set that stores the request number and the client address (M_{REQ}*'.num* and M_{REQ}*'.addr*) for all requests for which *atomic_dlv(*M_{REQ}*')* has already been called. If the server already received the request, the request is simply discarded (step 1). If several requests with the same command are delivered by the atomic multicast protocol, only the first one causes the execution of the command (step 2). When the command is executed, a reply is sent to the client.

This basic protocol makes at least one atomic multicast for each client request. This cost may be excessive depending on the rate of requests being issued. This cost can be greatly reduced using a *batching mechanism*, i.e., aggregating several requests in a single atomic multicast. The decision about batching requests is left for each server to take; if it assesses that the rate of requests is greater than a given bound, it starts collecting a number of requests before atomic multicasting them together in a single message. This mechanism introduces some delay in the system, so the client's T_{resend} has to take this delay into account.

4.2.1. Atomic Multicast Protocol The core of the algorithm executed by the servers is the atomic multicast protocol, which guarantees two properties: all correct servers deliver the same messages in the same order; if the sender is correct all servers deliver the sent message. A server is said to (atomically) multicast a message M if it calls *atomic_mcast(M)*, and it is said to (atomically) deliver a message M if *atomic_dlv(M)* is called in the server. The protocol is more formally defined in terms of four properties:

- *AM1 Validity.* If a correct server multicasts a message M with a vector with all MACs valid, then some correct server eventually delivers M.
- *AM2 Agreement.* If a correct server delivers a message M, then all correct servers eventually deliver M.
- *AM3 Integrity.* For any identifier ID, every correct server delivers at most one message M with identifier ID, and if *sender(M)* is correct then M was previously multicast by *sender(M)* [6].
- *AM4 Total order.* If two correct servers deliver two messages M_1 and M_2 then both servers deliver the two messages in the same order.

This definition is similar to other definitions found in the literature, e.g., in [10]. However, property AM1 does not guarantee that the message is delivered in case the message does not have a vector filled with valid MACs (i.e., MACs properly obtained using the key shared between the client and each of the servers). Recall that the objective of this vector of MACs if to prevent a malicious server from atomically multicasting a corrupted request (Section 4.1). Albeit the objective is to deal with malicious servers, if the client itself is malicious and sends a message with some invalid MACs, the message may not be delivered by the atomic multicast protocol.

The protocol is shown in Algorithm 2. It has four parts: initialization (lines 1-8), processing of a call to *atomic_mcast(M)* (lines 9-13), processing of the reception of an ACAST message (line 14), and a task that processes the messages stored in a number of buffers (lines 15-34). A correctness proof can be found in [6].

The protocol uses a single type of message:

$$\langle \text{ACAST, addr, mreq, msg\_id, sender\_eid, elist, threshold} \rangle$$

where *ACAST* is the message type, *addr* the address of the sender server, *mreq* the request message (*mreq* = M_{REQ}), *msg_id* a message number unique for the sender, *sender_eid* the eid of the server that atomically multicasted the message, *elist* is the list of *eid*'s of the processes involved in the protocol, and *threshold* is the value $\lfloor \frac{n-1}{2} \rfloor + 1$ (for $n = 2f + 1$, it is $f + 1$). The identifier of a message in property AM3 (ID) is: *(ACAST, msg_id, sender_eid, elist, threshold).*

Lines 1-7 initialize several local variables, including three sets used to store messages in different stages of processing: *Wait_tmo*, *Wait_thresh* and *Wait_deliv*. Line 8 starts task T1.

6 The predicate *sender(M)* gives the sender field of the header of M.

Algorithm 2 Atomic multicast protocol (server s_i).

```
INITIALIZATION:
 1: elist ← {all eid's of servers in $S$ in canonical order}
 2: msg_id_next ← 1                {number of next ACAST to send}
 3: threshold ← $\lfloor \frac{n-1}{2} \rfloor + 1$ {threshold for TMO service $(f+1)$}
 4: order_next ← 1                 {number of next request to deliver}
 5: Wait_tmo ← ∅ {set w/recvd ACASTs while TMO unknown}
 6: Wait_thresh ← ∅ {set w/ACASTs while thresh. not reached}
 7: Wait_deliv ← ∅          {set with requests waiting for delivery}
 8: activate task (T1)

WHEN ATOMIC_MCAST($M_{REQ}$) IS CALLED DO
 9: if verify_mac($M_{REQ}$.vec[$s_i$]) then
10:   multicast $M_{ACAST}$ = ⟨ACAST, addr$_i$, $M_{REQ}$,
      msg_id_next, my_eid, elist, threshold⟩ to servers S \ {$s_i$}
11:   err, tag ← TTCB_TMO_send(eid$_i$, elist, threshold,
      msg_id_next, Hash($M_{REQ}$))
12:   msg_id_next ← msg_id_next + 1
13:   Wait_thresh ← Wait_thresh ∪ {($M_{ACAST}$,tag)}

WHEN $M_{ACAST}$ IS RECEIVED DO
14: Wait_tmo ← Wait_tmo ∪ {$M_{ACAST}$}

TASK T1:
15: loop
16:   for all $M_{ACAST}$ ∈ Wait_tmo do {msgs w/unknown TMO}
17:     if verify_mac($M_{REQ}$.vec[s$_i$]) then hash ← Hash(
        $M_{ACAST}$.mreq) else hash ← ⊥
18:     err, tag ← TTCB_TMO_receive(eid$_i$, $M_{ACAST}$.elist,
        $M_{ACAST}$.threshold, $M_{ACAST}$.msg_id, hash, $M_{ACAST}$.
        sender_eid)
19:     if err ≠ TMO_UNKNOWN then
20:       Wait_tmo ← Wait_tmo \ {$M_{ACAST}$}
21:       if (err ≠ WRONG_HASH) or (hash = ⊥) then
22:         Wait_thresh ← Wait_thresh ∪ {($M_{ACAST}$,tag)}
23:   for all ($M_{ACAST}$,tag)∈Wait_thresh do {thresh not reachd}
24:     err, n, hash, prop_mask ← TTCB_TMO_decide(tag)
25:     if err ≠ THRESHOLD_NOT_REACHED then
26:       Wait_thresh ← Wait_thresh \ {($M_{ACAST}$,tag)}
27:       if Hash($M_{ACAST}$.mreq) = hash then
28:         Wait_deliv ← Wait_deliv ∪ {($M_{ACAST}$.mreq,n)}
29:         if $M_{ACAST}$.addr ≠ addr$_i$ then {if not the sender}
30:           multicast $M_{ACAST}$ to {$\forall_{s_j \in S}$ : $s_j$ ∉prop_mask}
31:   while $\exists_{(M_{REQ},n) \in Wait\_deliv}$ : n = order_next do
      {messages waiting to be delivered}
32:     Wait_deliv ← Wait_deliv \ {($M_{REQ}$,n)}
33:     order_next ← order_next + 1
34:     ATOMIC_DLV($M_{REQ}$)
```

When *atomic_mcast(M$_{REQ}$)* is called, the server calls *verify_mac* to test if the MAC that corresponds to itself (s_i) in the vector of MACs is valid (line 9). If it is not, the server simply dismisses the message. If the MAC is valid, the request M$_{REQ}$ is enveloped in an ACAST message and multicasted to all servers except the sender (line 10). Then, the server starts the execution of one instance of the TTCB TMO service by calling TTCB_TMO_send (line 11). Each call to *atomic_mcast* causes at most one execution of the TMO service. After starting the TMO service, the server puts the ACAST message in the set *Wait_thresh*, waiting for the TMO threshold to be achieved (line 13). When an ACAST message is received by a server, it is simply stored in *Wait_tmo* (line 14).

Task T1 is permanently checking if the messages in the three sets can be processed. Messages in *Wait_tmo* are handled in lines 16-22. For each message in *Wait_tmo*, task T1 makes a call to TTCB_TMO_receive (lines 16-18). If the MAC corresponding to s_i is valid, the hash of message is given to TTCB_TMO_receive (lines 17-18). Otherwise, a value out of the range of valid hashes is given, $\perp$ (lines 17-18). If the local TTCB is still not aware of that TMO execution[7], then TTCB_TMO_receive returns the error TMO_UNKNOWN. If the TTCB is aware of the TMO but the hash of the request is wrong, then an error WRONG_HASH is returned. If the TTCB is aware of the TMO and either the hash is correct, or the hash is $\perp$ (the MAC is invalid), the message is removed from *Wait_tmo* and inserted in *Wait_thresh* (lines 19-22). If the TTCB is aware of the TMO but the hash is wrong (but not $\perp$), the message is discarded since it has been corrupted at some stage (lines 19-22).

The set *Wait_thresh* contains messages waiting for the number of calls to their TMO to reach the threshold. These messages are handled in lines 23-30. The purpose of the *threshold* is to guarantee that the servers only decide to deliver a message if they eventually become able to deliver it. In other words, they can only decide to deliver a message if at least one correct server has the message. This is guaranteed if at least $f + 1$ servers prove that they know the hash of the message, therefore the threshold is set to $f + 1$ (line 3). Notice that a server that received a message with an invalid MAC does not contribute to the threshold, since it gives TTCB_TMO_receive the value $\perp$ instead of the hash of the message (lines 17-18). When the threshold is reached, the message is removed from *Wait_thresh* (lines 25-26). If the message corresponds to the hash returned, the message is inserted in *Wait_deliv* (lines 27-28). Then, if the server is not the message sender, it resends the message to the servers that did not 'contribute' to the threshold, i.e., to the servers not in *prop_mask* (lines 29-30). The rationale for resending the message is that a malicious sender can send the message only to a subset of the servers; therefore, these servers may not have the message.

The set *Wait_deliv* keeps messages that already have an order number assigned by the TMO service, therefore they can be delivered. These messages are handled in lines 31-34. The algorithm keeps a number with the next message to be delivered, *order_next*. If the next message to be delivered is stored in *Wait_deliv*, then task T1 delivers it (lines 31-34). Otherwise, the message has to wait for its turn.

4.2.2. FLP Impossibility Result The consensus problem has been proven to be impossible to solve deterministically in asynchronous systems if a process is allowed to fail, even if only by crashing [9]. This FLP impossibility result also applies to the atomic multicast problem since it is essentially equivalent to consensus [10]. Our system is not bound by FLP, since it is not fully asynchronous: it is mostly asynchronous, but includes the TTCB subsystem, which is synchronous. The problem of atomic multicast is essentially equivalent to a consensus about the set of messages to deliver and their order. Our protocol leaves this consensus to the TTCB TMO service, which is executed in a synchronous environment, therefore FLP does not apply.

5. Performance

The evaluation of the performance of distributed protocols is usually made in terms of time and message complexities. In asynchronous systems, the *time complexity* is usually measured in terms of the maximum number of *asynchronous rounds* of message exchange. An asynchronous round involves a process sending a message and receiving one or more messages in response. For the Byzantine fault model, only the number of rounds executed by correct processes matter, since malicious processes can behave arbitrarily. We consider separately the number of rounds of TMO execution.

The time complexity is two rounds of message exchange in the payload network, plus one round of TMO executions. Let us justify this complexity by presenting the worst case. The client sends a request to a sender s_j (half round), but s_j is crashed (or is malicious), so s_j does not multicast the message to the other servers. This situation forces the client to resend the request to another f servers, which we count as another half round. Then, all correct servers that received the request, multicast the request in an ACAST message to all other servers (half round) and start one TMO (one round of TMO executions, since all TMOs are executed in parallel). When the first of these TMOs terminates, the com-

7 The TMO is started in the local TTCB of the server that atomically multicasts the ACAST message, so the information about the TMO takes a certain time to be broadcasted and received by the other local TTCBs. Therefore, it is not possible to guarantee that the TMO information will be available in a local TTCB when the corresponding ACAST message is received.

mand is executed and all correct servers send a reply to the client (half round). Therefore, there are two rounds of message exchange plus one round of TMO executions.

The *message complexity* is measured in number of messages (unicasts) sent. We start by discussing this complexity when the batching mechanism is disabled. The complexity of the SMR service can be divided essentially in three cases:

1. *One request.* For each command, a client sends only one REQUEST message to a single server because the client is correct, the server is correct, and the servers answer in less than T_{resend} units of time measured in the client's clock. A single TMO is executed.
2. $f+1$ *requests.* For each command, a client sends REQUEST messages to $f+1$ servers because the servers do not respond before T_{resend}, although both the client and the server for which it first sends the request are correct. $f+1$ TMOs are executed.
3. n *requests.* For each command, a malicious client sends REQUEST messages to all n servers. n TMOs are executed. A malicious client can issue any number of commands but the SMR protocol prevents it from forcing the execution of more than n TMOs by command (see Algorithm 1).

Requests	Message complexity	TMOs
1	$O(n^2)$	1
$f+1$	$O(n^3)$	$f+1$
n	$O(n^3)$	n

Table 1. Message complexity and number of TMOs executed (batching disabled).

Table 1 summarizes the message complexities for the three situations. The deduction of these values is straightforward. The table assumes the batching mechanism is disabled. However, the purpose of this mechanism is precisely to reduce these values. If we consider that the average number of requests batched in each atomic multicast is B, then the message complexities and the number of TMOs presented in the table have to be divided by B. Therefore, the higher the value of B, the higher the reduction in the complexity and number of TMOs. Nevertheless, there is a tradeoff. To increase B the algorithm has to delay requests until a certain number can be batched in an atomic multicast, therefore increasing the average latency of the algorithm.

6. Related Work

The state machine approach was first introduced by Lamport for systems in which faults were assumed not to occur [13]. Later, Schneider generalized the approach for systems with crash faults [20]. More recently, two Byzantine-resilient state machine replication systems with resilience $\lfloor \frac{n-1}{3} \rfloor$ appeared: Rampart and BFT.

Rampart is an intrusion-tolerant group communication system. It provides a set of communication primitives and a membership service, which handles the joining and leaving of group members [19]. When a message is atomically multicast to the group, a reliable multicast protocol is used to send the message. Then, a special process, the sequencer, defines an order for the messages and also reliably multicasts this order to the group. All these protocols use digital signatures to authenticate some messages [15]. Rampart is mostly asynchronous but assumes enough synchrony to detect process failures. Replication is implemented by a set of servers, which form a group [19]. Clients send their requests to a server of their choice, similarly to our algorithm. The output of the service has to be voted so that the results provided by correct servers prevail over those returned by malicious servers. Two solutions were implemented: one very similar to ours, and another one based on a (k,n)-threshold signature scheme, which has poor performance. Besides Rampart, there are two other intrusion-tolerant group communication systems: SecureRing [11] and SecureGroup [16]. However, there is no discussion about their use for the implementation of the state machine approach. The resilience is the same.

BFT is a Byzantine-resilient state machine replication service. The system is optimized for having good performance, therefore, on the contrary to Rampart, most of the time it does not use public-key cryptography. In BFT, all clients send the requests to the same server, the primary. Then, the primary atomically multicasts the request to the backups (the other servers); all replicas execute the request and send the result to the client; the client waits for $f+1$ replies with the same result, which is the result of the operation. BFT assumes enough synchrony to detect the failure of the primary. When it fails, a new primary is elected.

SINTRA provides a number of group communication primitives that can be used to support SMR [2]. These primitives are based on a randomized Byzantine agreement protocol, therefore they are strictly asynchronous. The resilience is also $\lfloor \frac{n-1}{3} \rfloor$.

FS-NewTOP is a recent intrusion-tolerant SMR system based on fail-signal (FS) processes, i.e., processes that announce when they fail [17]. Each FS process is implemented by two nodes connected by a synchronous channel. Each node monitors its peer. When one node detects that its peer has misbehaved in some way, it signals the failure to all processes and stops the FS process. The resilience is allegedly $4f+2$, which is sub-optimal. However, the algorithm does not tolerate the failure of two nodes, if they are part of the same FS process.

Pedone et al. used *weak ordering oracles* to solve crash-tolerant agreement problems in asynchronous systems [18].

The oracle gives a hint about the order of the messages, which may be right or wrong. The hint is simply the order in which the messages are received from the network, which is often right in a LAN. Our 'TTCB with TMO' oracle might be considered to be a perfect ordering oracle.

7. Conclusion

This paper proposes a novel state machine approach solution. The algorithm is executed in an asynchronous and Byzantine environment, with the exception of a synchronous and secure distributed subsystem, the Trusted Timely Computing Base wormhole. The algorithm is based on a novel TTCB service, the Trusted Multicast Ordering service, which defines an order for a set of messages represented by their hashes. Using this service, we managed to design an atomic multicast protocol with a resilience lower than the maximum theoretical bound in asynchronous systems: $\lfloor \frac{n-1}{2} \rfloor$ against $\lfloor \frac{n-1}{3} \rfloor$. The paper also shows how the TTCB can be used to circumvent FLP. The performance of the system was assessed in terms of time and message complexities, and number of TMOs executed. The system is currently being implemented.

References

[1] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, Oct. 1985.

[2] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 167–176, June 2002.

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

[4] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper. In *Real-Time Linux Workshop*, Nov. 2000.

[5] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 2–11, Oct. 2002.

[6] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. DI/FCUL TR 04–6, Department of Informatics, University of Lisbon, July 2004.

[7] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel (extended version). DI/FCUL TR 01–12, Department of Computer Science, University of Lisbon, 2001.

[8] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252, Oct. 2002.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[10] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[11] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, Nov. 2001.

[12] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, Jan. 2003.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[15] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[16] L. E. Moser, P. M. Melliar-Smith, and N. Narasimhan. The SecureGroup communication system. In *Proceedings of the IEEE Information Survivability Conference*, pages 507–516, Jan. 2000.

[17] 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 International Conference on Dependable Systems and Networks*, pages 227–236, June 2003.

[18] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 44–61, Oct. 2002.

[19] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 1995.

[20] F. B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, Apr. 1982.

[21] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[22] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. Springer-Verlag, 2003.

[23] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, Aug. 2002.

[24] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag, 2003.