Efficient Byzantine-Resilient Reliable Multicast on a Hybrid Failure Model[*]

Miguel Correia **Lau Cheuk Lung** **Nuno Ferreira Neves** **Paulo Veríssimo**
Faculdade de Ciências da Universidade de Lisboa
Bloco C5, Campo Grande, 1749-016 Lisboa - Portugal
{mpc,lau,nuno,pjv}@di.fc.ul.pt

Abstract

The paper presents a new reliable multicast protocol that tolerates arbitrary faults, including Byzantine faults. This protocol is developed using a novel way of designing secure protocols which is based on a well-founded hybrid failure model. Despite our claim of arbitrary failure resilience, the protocol needs not necessarily incur the cost of "Byzantine agreement", in number of participants and round/message complexity. It can rely on the existence of a simple distributed security kernel – the TTCB – where the participants only execute crucial parts of the protocol operation, under the protection of a crash failure model. Otherwise, participants follow an arbitrary failure model.

The TTCB provides only a few basic services, which allow our protocol to have an efficiency similar to that of accidental fault-tolerant protocols: for f faults, our protocol requires f+2 processes, instead of 3f+1 in Byzantine systems. Besides, the TTCB (which is synchronous) allows secure operation of timed protocols, despite the unpredictable time behavior of the environment (possibly due to attacks on timing assumptions).

1 Introduction

Protocols that are able to tolerate Byzantine faults have been extensively studied in the past 20 years [9, 17], and they have been applied to a number of well-known problems, such as consensus and group communication primitives with different order guarantees. These protocols are usually built for a system composed by a set of cooperating processes (or machines) interconnected by a network. The processes may fail arbitrarily, e.g., they can crash, delay or not transmit some messages, generate messages inconsistent with the protocol, or collude with other faulty processes with malicious intent. The synchrony assumptions about the network and process execution have either been the synchronous or the asynchronous models. Recent research in this area, however, has mostly focused on asynchronous systems, since this model is well-suited for describing networks like the Internet and other WANs with unpredictable timeliness (examples can be found in [4, 18, 10, 8, 14, 3]). The assumption of this model has also one added advantage – the resulting protocol tolerates timing attacks.

Nevertheless, the asynchronous model has some drawbacks, and among them is the constrain that it imposes on the maximum number of processes that are allowed to fail simultaneously. For instance, Bracha and Toueg showed that, assuming Byzantine faults, it is impossible to send reliable multicasts if there are more than $f = \frac{n-1}{3}$ faulty processes in a system with n processes [2]. Their proof was valid even under strong assumptions about the network, such as the availability of reliable authenticated channels. The main problem with these constraints is that they are difficult or impossible to substantiate in practical systems, since malicious faults (attacks and intrusions) may be performed by intelligent entities. If the machines contain a set of common vulnerabilities, it is quite feasible to build a program that is able to attack and compromise a large number of nodes in a short time.

This paper describes a new reliable multicast protocol for asynchronous systems with a hybrid failure model. The basic idea of this type of model is to make distinct failure assumptions about different components of the system, ranging from arbitrary to fail-controlled. In our case, processes and network can behave in a Byzantine way, however, we assume the existence of a distributed security kernel that can only fail by crashing. This kernel only provides limited functionality, but can be called by processes to execute a few small steps of the protocol. By relying on this kernel, our protocol is highly efficient, for instance in terms of

[*]This work was partially supported by the EC, through project IST-1999-11583 (MAFTIA), and by the FCT, through the Large-Scale Informatics Systems Laboratory (LASIGE) and the project POSI/1999/CHS/33996 (DEFEATS).

message complexity, when compared with traditional protocols. Moreover, it imposes constraints on the number of process failures that are similar to accidental fault-tolerant protocols: for f faults, our protocol requires $n \geq f + 2$ processes, instead of $n \geq 3f + 1$. In reality, our protocol does not impose a minimum number of correct processes. However, in practice, we say that the number of processes has to be $n \geq f + 2$ to denote the notion that the problem is vacuous if there are less than two correct processes. This was already pointed out by Lamport et al. [9].

The design and implementation of the security kernel that is being considered, the Trusted Timely Computing Base (TTCB), has been presented elsewhere [5]. Since this kernel is both secure and timely, it can offer security and time related services (see Section 2). These services can be utilized in a useful way by the processes because, by construction, the TTCB was implemented with the following fundamental objectives in mind: 1) the TTCB is a distributed component, with limited services and functionality, that resides inside potentially insecure hosts, yet is and remains reliable, secure and timely; and 2) it is possible to ensure correct –reliable, secure, timely– interactions between processes in the host, and that component.

The paper makes the following two main contributions:

- It presents a novel way of designing asynchronous Byzantine-resilient protocols, which rely on a distributed security kernel to execute a few crucial steps.
- It describes a new reliable multicast protocol which is highly efficient and imposes constraints on the number of faulty processes in the order of $n \geq f + 2$.

2 System Model and the TTCB

Figure 1 presents the architecture of the system. Each host contains the typical software layers, such as the operating system and runtime environments, and an extra component, the TTCB. The TTCB is a distributed entity with local parts in the hosts and a control channel. The local parts, or *local TTCBs*, are computational components with activity, conceptually separate from the hosts' operating system. The *control channel* is a private communication channel or network that interconnects the local TTCBs. It is conceptually separated from the payload network, the network used by the hosts to communicate. With the exception of the TTCB, the whole system is assumed to be asynchronous. We can make working assumptions on message delivery delays; they may even hold many times; we can (and will) use them to ensure system progress; but we can *never* assume that bounds are known or even exist, for message delivery or for the interactions between a process and the local TTCB.

The following sections describe the TTCB respectively from the point of view of security and timeliness.

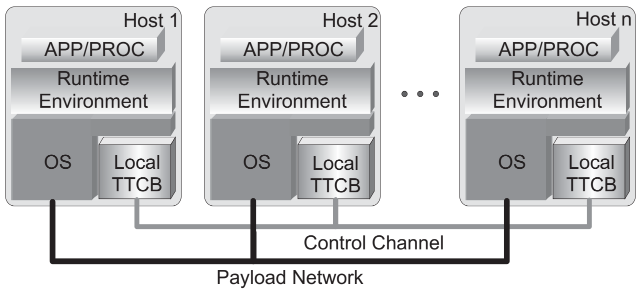

Figure 1. System architecture with a TTCB.

2.1 TTCB and Security

Most of the research and engineering on security aims at making complete systems secure, or at building concepts and mechanisms useful to reach this goal. The construction of a secure system is a hard task as the constant news of successful attacks show. In our work with the TTCB, however, we have shown that it is feasible to build a secure distributed component with limited functionality [5].

The idea of designing protocols with the assistance of a secure component is novel, and relies on the concept of *hybrid failure assumptions*. Fault-tolerant systems are usually built using either arbitrary or controlled failure assumptions. Arbitrary failure assumptions consider that components can fail in any way, although in practice constraints have to be made. These assumptions are specially adequate for systems with malicious faults –attacks and intrusions [16]– since these faults are induced by intelligent entities, which are hard to restrict and model. Byzantine protocols follow this type of assumption since they consider that processes can fail arbitrarily, although they put limits on the number of processes that are allowed to fail. Controlled failure assumptions are used for instance in systems where components can only fail by crashing. Hybrid failure assumptions bring together these two worlds: some components are assumed to fail in a controlled way, while others may fail arbitrarily. The TTCB is suited for this type of systems because it only fails by crashing, and the rest of the system, e.g., the payload network, processes, and operating systems, can follow an arbitrary failure model.

The protocol uses only two TTCB's *security-related services*. The *Local Authentication service* allows processes to communicate securely with the TTCB. The service authenticates the local TTCB before the process, and establishes a shared symmetric key between both [5].

The *Trusted Block Agreement service* is the main building block for secure protocols. It delivers a value obtained from the agreement of values proposed by a set of processes. The values are blocks with limited size, so this service cannot be used to make all agreement related operations of the system, but only to do critical steps of the protocols.

The agreement service (for short) is formally defined in terms of three functions: *TTCB_propose*, *TTCB_decide* and *decision*. A process *proposes a value* when it calls *TTCB_propose*. A process *decides a result* when it calls *TTCB_decide* and receives a result. The function *decision* defines how the result is calculated in terms of the inputs to the service. The value is a "small" block of data with fixed length (currently 160 bits). The *result* is composed by a value and two masks with one bit per process involved in the service. Formally, the agreement service is defined by the following properties:

- *Termination.* Every correct process eventually decides a result.
- *Integrity.* Every correct process decides at most one result.
- *Agreement.* If a correct process decides *result*, then all correct processes eventually decide *result*.
- *Validity.* If a correct process decides *result* then *result* is obtained by applying function *decision* to the values proposed.
- *Timeliness.* Given an instant $tstart$ and a known constant $T_{agreement}$, a process can decide by $tstart + T_{agreement}$.

It is assumed that a correct process is capable of securely calling the interface of the local TTCB, i.e., an attacker can not systematically intercept, delay or substitute without detection the information exchanged between the process and the local TTCB (see discussion is Section 2.3). The last property, *Timeliness*, only guarantees that by $tstart + T_{agreement}$ the decision is available at the local TTCB. Since the environment is asynchronous, it will probably take longer for the process to obtain the result (a call to function *TTCB_decide* can be arbitrarily delayed).

The interface of the agreement service has two functions:

```
out ←TTCB_propose(eid, elist, tstart, decision, value)
result ←TTCB_decide(eid, tag)
```

A process calls *TTCB_propose* to propose its value. eid is the unique identification of a process before the TTCB, obtained using the Local Authentication Service. $elist$ is a list with the eids of the processes involved in the agreement. $tstart$ is a timestamp with the following objective. Ideally the agreement should be executed when all processes in $elist$ proposed their value. However, if the service was to wait for all processes to propose, a malicious process would be able to postpone the service execution eternally simply by not proposing its value. The purpose of $tstart$ is to avoid this problem: when all processes proposed, the service starts; however, if the service is not initiated by $tstart$, then it starts at that instant and no more proposals are accepted. A proposal made after $tstart$ is rejected and an error is returned. *decision* indicates the function that should be used to calculate the value that will be decided (the TTCB offers a limited set). *value* is the value proposed. Function *TTCB_propose* returns a structure out with two fields: *out.error* is an error code and *out.tag* is a unique identifier of the execution of the agreement. The TTCB knows that two calls to *propose* made by different processes pertain to the same agreement execution if they have the same value for *(elist, tstart, decision)*.

Processes call *TTCB_decide* to get the result of the agreement. tag is the unique identifier returned by *TTCB_propose*, and is used to specify the agreement instance. *result* is a record with four fields: *result.error* gives an error code; *result.value* is the value decided; *result.proposed-ok* is a mask with one bit per process in $elist$, where each bit indicates if the corresponding process proposed the value that was decided or not; *result.proposed-any* is a similar mask but that indicates which processes proposed any value.

2.2 TTCB and Time

From the point of view of time, the objective of the TTCB is to support *systems with partial-synchronous models*. Research in distributed systems has traditionally been divided between two canonical models: fully synchronous and fully asynchronous. Partial-synchronous models try to give the best of both worlds, allowing timeliness specifications but accepting that they can fail [22, 6].

The TTCB provides a set of time services whose main objective is precisely to *detect the failure of timeliness specifications*. This is only possible because the TTCB is timely, i.e., the TTCB is a real-time (synchronous) component. Many of the time related ideas and services of the TTCB were based on the work of the Timely Computing Base [23].

The protocol presented in this paper uses a single time service – the Trusted Absolute Timestamping Service. This service provides globally meaningful timestamps. It is possible to obtain timestamps with this characteristic because local TTCBs clocks are synchronized.

2.3 Processes and Failures

A process is *correct* if it always follows the protocol until the protocol completion. There are several circumstances, however, that might lead to a process failure. In an arbitrary failure model, which is the model being considered in this paper, no restrictions are imposed on process failures, i.e., they can fail arbitrarily. A process can simply stop working, or it can send messages without regard of the protocol, delay or send contradictory messages, or even collude with other malicious processes with the objective of breaking the protocol. In the rest of this section we will look into a few examples of attacks that are specific to our architecture, and that might lead to the failure of the corresponding process.

A personification attack can be made by a local adversary if it is able to get the pair $(eid, secret)$, which lets a process communicate securely with the local TTCB. Before a process starts to use the TTCB, it needs to call the Local Authentication Service to establish a secure channel with the local TTCB. The outcome of the execution of this procedure is a pair $(eid, secret)$, where eid is the identifier of the process and $secret$ is a symmetric key shared with the local TTCB. If an attacker penetrates a host and obtains this pair, it can impersonate the process before the TTCB and the TTCB before the process.

Another personification attack is possible if the attacker obtains the symmetric keys that a process shares with other processes. In this case, the attacker can forge some of the messages sent between processes. Most of the messages transmitted by the protocol being proposed do not need to be authenticated and integrity protected because corruptions and forgeries can be detected with the help of the TTCB. The only exception happens with the acknowledgments sent by the protocol, where it is necessary to add a vector of message authentication codes. A successful attack to a host and subsequent disclosure of the shared keys of a process, allows an attacker to falsify some acknowledgements. If the keys can be kept secret, then he or she can only disrupt or delay the communication, in the host or the network.

A denial of service attack happens if an attacker prevents a process from exchanging data with other processes by systematically disrupting or delaying the communication. In asynchronous protocols typically it is assumed that messages are eventually received (reliable channels), and when this happens the protocol is able to make progress. To implement this behavior processes are required to maintain a copy of each message and to keep re-transmitting until an acknowledgement arrives (which might take a long time, depending on the failure). In this paper we decided to take a different approach: if an attacker can systematically disrupt the communication of a process, then the process is considered failed as soon as possible, otherwise the attacker will probably disturb the communication long enough for the protocol to become useless. For example, if the payment system of an e-store is attacked and an attempt of paying an item takes 10 hours (or 10 days) to proceed, that is equivalent to a failure of the store.

In channels with only accidental faults it is usually considered that no more than Od messages are corrupted/lost in a reference interval of time. Od is the *omission degree* and tests can be made in concrete networks to determine Od with any desired probability [24]. If a process does not receive a message after $Od + 1$ retransmissions from the sender, with Od computed considering only accidental faults, then it is reasonable to assume that either the process crashed, or an attack is under way. In any case, we will consider the receiver process as failed. The reader, however, should notice that Od is just a parameter of the protocol. If Od is set to a very high value, then our protocol will start to behave like the protocols that assume reliable channels.

Note that the omission degree technique lies on a synchrony hypothesis: we 'detect' omissions if a message does not arrive after a timeout longer than the 'worst-case delivery delay' (the hypothesis). Furthermore, we 'detect' crash if the omission degree is exceeded. In our environment (since it is asynchronous, bursts of messages may be over-delayed, instead of lost) this artificial hypothesis leads to forcing the crash of live but slow (or slowly connected) processes. There is nothing wrong with this, since it allows progress of the protocol, but this method is subject to inconsistencies if failures are not detected correctly. In our system, we rely on the timing failure detector of the TTCB to ensure complete and accurate failure detection amongst all participants [23], and feed a membership service complementing the reliable multicast protocol being described. These mechanisms are out of the scope of the present paper, but substantiate the correctness of the omission degree technique for asynchronous environments.

3 Protocol Definition and Properties

In each execution of a multicast there is one sender process and several recipient processes. A message transmitted to a group should be delivered to all member processes (with the limitations mentioned below), including the sender. No assurances, however, are provided about the order of message delivery. Each process can deliver its messages in a distinct order. In the rest of the paper, we will make the classical separation of *receiving* a message from the network and *delivering* a message – the result of the protocol execution.

Informally, a reliable multicast protocol enforces the following [2]: 1) all correct processes deliver the same messages, and 2) if a correct sender transmits a message then all correct processes deliver this message. These rules do not imply any guarantees of delivery in case of a malicious sender. However, one of two things will happen, either the correct processes never complete the protocol execution and no message is ever delivered, or if they terminate, then they will all deliver the same message. No assumptions are made about the behavior of the malicious (recipient) processes. They might decide to deliver the correct message, a distinct message or no message.

Formally, a reliable multicast protocol has the properties below [7]. The predicate $sender(M)$ gives the message field with the sender, and $group(M)$ gives the "group" of processes involved, i.e., the sender and the recipients (note

BRM-M Sender and Recipient protocol

```
1  // ——— Phase 1 ———
2  if I am the sender then                          // SENDER process
3      M := (DAT, my-eid, elist, TTCB_getTimestamp() + $T_1$, data);
4      multicast M to elist except sender; n-sends := 1;
5  else                                             // RECIPIENT processes
6      read_blocking(M); n-sends := 0;
7  propose := TTCB_propose(M.elist, M.tstart,
       TTCB_TBA_RMULTICAST, H(M));
8  do decide:=TTCB_decide(propose.tag);
       while (decide.error$\neq$TTCB_TBA_ENDED);
9  if (decide.proposed-ok contains all recipients) then deliver M; return;
10 // ——— Phase 2 ———
11 M-deliver := $\perp$;
12 mac-vector := calculate macs of (ACK, my-eid, M.elist, M.tstart,
       decide.value);
13 M-ack := (ACK, my-eid, M.elist, M.tstart, mac-vector);
14 n-acks := 0; ack-set := eids in decide.proposed-ok;
15 t-resend := TTCB_getTimestamp();
16 do
17     if (M.type = DAT) and (H(M) = decide.value) then
18         M-deliver := M;
19         ack-set := ack-set $\cup$ {my-eid};
20         if (my-eid $\notin$ decide.proposed-ok) and (n-acks $<$ Od+1) then
21             multicast M-ack to elist except my-eid; n-acks := n-acks + 1;
22     else if (M.type = ACK) and (M.mac-vector[my-eid] is ok) then
23         ack-set := ack-set $\cup$ {M.sender};
24     if (M-deliver $\neq \perp$) and (TTCB_getTimestamp() $\geq$ t- resend) then
25         multicast M-deliver to elist except (sender and eids in ack- set);
26         t-resend := t-resend + Tresend; n-sends := n-sends + 1;
27     read_non_blocking(M); // sets M = $\perp$ if no messages to be read
28 while (ack-set does not contain all recipients) and (n-sends $<$ Od+1);
29 deliver(M-deliver);
```

Figure 2. BRM-M protocol.

that we consider that the sender also delivers).

- *Validity:* If a correct process multicasts a message M, then some correct process in $group(M)$ eventually delivers M.
- *Agreement:* If a correct process delivers a message M, then all correct processes in $group(M)$ eventually deliver M.
- *Integrity:* For any message M, every correct process p delivers M at most once and only if p is in $group(M)$, and if $sender(M)$ is correct then M was previously multicast by $sender(M)$.

4 The BRM-M Protocol

The Byzantine Reliable Multicast BRM-M protocol is executed in two phases. In the first, the sender multicasts the message one time for the recipients, and then it securely transmits a hash code through the TTCB agreement service. This hash code is used by the receivers to ensure the integrity and authenticity of the message. If there are no attacks and no congestion in the network, with high probability the message is received by all recipients, and the protocol can terminate immediately. Otherwise, it is necessary to enter the second phase. Here, processes retransmit the message until either a confirmation arrives or the $Od + 1$ limit is reached.

Figure 2 shows an implementation of the protocol. A message consists of a tuple with the following fields $(type, sender, elist, tstart, data)$. $type$ indicates if it is a data message (DAT) or an acknowledgement (ACK). $sender$ is the identifier of the sender process, and $data$ is either the information given by the application or a vector of MACs (see below). $elist$ is a list of eid's with the format accepted by the TTCB agreement service. The first element of the list is the eid of the sender, the others are the eid of the receivers. $tstart$ is the timestamp that will be given to the agreement service. Each execution of the protocol is identified by $(elist, tstart)$. The protocol uses two low level read primitives, one that only returns when a new message is available, $read\_blocking()$, and another that returns immediately either with a new message or with a non-valid value ($\perp$) to indicate that no message exists, $read\_non\_blocking()$. These two primitives only read messages with the same value of $(elist, tstart)$ which correspond to a given instance of the protocol execution. Other values of the pair are processed by other instances of the protocol. We assume that there is a garbage collector that throws away messages for instances of the protocol that have already finished running (e.g., delayed message retransmissions). This garbage collector can be constructed by keeping in a list the identifiers of the messages already delivered and comparing these with the arriving messages.

With the exception of the beginning, the code presented in the figure is common both to the sender and the recipients. If the process is a sender, it constructs and multicasts the message to the receivers (lines 3-4). $tstart$ is set to the current time plus a delay T_1. T_1 should be proportional to the average message transmission time, i.e., it should be calculated in such a way that there is a reasonable probability of message arrival before $tstart$.

Recipient processes start by blocking, waiting for a message arrival (line 6). Depending on whether there are or not message losses, the received message might be of type DAT or ACK, or a corrupted message with the fields $(elist, tstart)$ correct. The variable *n-sends* contains the number of messages that were multicast (lines 4 and 6). Next, both sender and recipients propose the hash of the message, $H(M)$, to the agreement service (M is the message transmitted by the sender, or the first message received by the recipient), and then they block waiting for the result of the agreement (line 7-8). The decision function used by the protocol, $TTCB\_TBA\_RMULTICAST$, selects as result the value proposed by the first process in $elist$, which in our case is the sender. A hash function is basically a one-way function that compresses its input and produces a fixed sized digest (e.g., 128 bits for MD5). We assume that

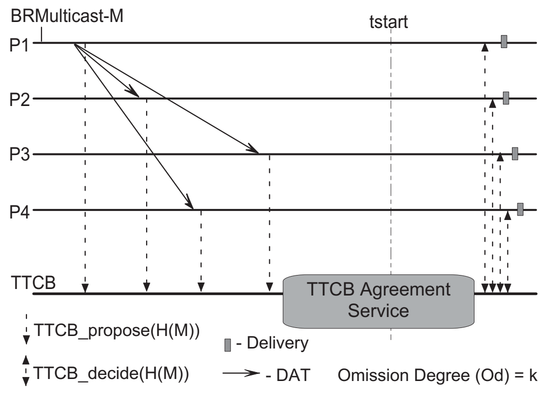

Figure 3. Protocol execution (best case).

an attacker is unable to subvert the cryptographic properties of the hash function, such as weak and strong collision resistance [12]. Since the system is asynchronous, there is always the possibility, although highly improbable, that the sender experiences some delay and it tries to propose after $tstart$. In this case, $TTCB\_propose$ will return the error $TTCB\_TSTART\_EXPIRED$ and the sender process should abort the multicast, and the application can retry the multicast later (for simplicity this condition is omitted from the code). If all processes proposed the same hash of the message, all can deliver and terminate (line 9). Recall that field *proposed-ok* indicates which processes proposed the same value as the one that was decided, i.e., $H(M)$. Figure 3 illustrates an execution where processes terminate after this first phase of the protocol.

The second phase is executed if for some reason one or more processes did not propose the hash of the correct message by $tstart$. Variable *M- deliver* is used to store the message that should be delivered, and is initialized to a value outside the range of messages (line 11). The protocol utilizes message authentication codes (MAC) to protect ACK messages from forgery [12]. This type of signature is based on symmetric cryptography, which requires a different secret key to be shared between every pair of processes. Even though, MACs are not as powerful as signatures based on asymmetric cryptography, they are sufficient for our needs, and more importantly, they are several orders of magnitude faster to calculate. Since ACKs are multicast to all processes, an ACK does not take a single MAC but a vector of MACs, one per each pair (sender of ACK, other process in $elist$) [4]. A MAC protects the information contained in the tuple *(ACK, my-eid, M.elist, M.tstart, decide.value)*, and is generated using the secret key shared between each pair of processes (lines 12-13). Next, processes initialize variables *n-ack* and *ack-set* (line 14). The first one will count the number of ACKs that have been sent. The second one will store the eid of the processes that have already confirmed the reception of the message, either by proposing the correct $H(M)$ to the agreement (line 14) or with an acknowledge message. *t-resend* indicates the instant when the next retransmission should be done (line 15). It is initialized to the current time, which means that there will be retransmission as soon as possible.

The loop basically processes the arriving messages (lines 17-23), does the periodic retransmissions (lines 24-26), and reads new messages (line 27). The loop goes on until $Od+1$ messages are sent or all recipients acknowledged the reception of the message (line 28).

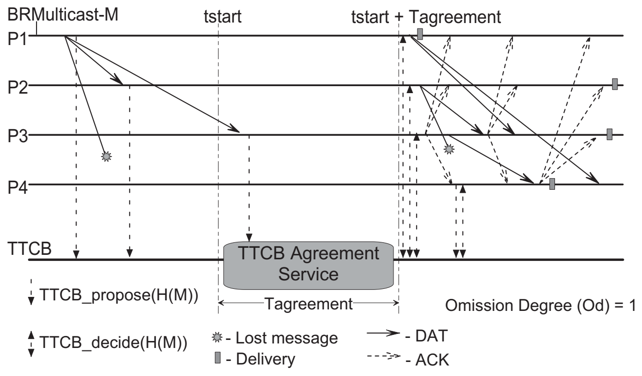

Figure 4. Protocol execution.

Figure 4 represents an execution of the protocol. The sender multicasts the message once, P2 receives it in time to propose $H(M)$, P3 receives the message late and P4 does not receive. When the agreement terminates all processes except P4 have the message and get the result from the TTCB (P4 does not even know that the protocol is being executed). At this point, by observing the result of the agreement, all become aware that only P1 and P2 proposed the hash. Therefore, both P1 and P2 multicast the message to P3 and P4. P3 multicasts an ACK to all processes confirming the reception and sends the message to P4. P1 terminates at this moment because it has already sent the message $Od + 1$ times. The first message P4 receives is the ACK sent by P3. P4 saves it in *ack-set* and gets the result of the agreement. Then it receives the right message, and multicasts an ACK. At this moment all processes terminate.

5 Protocol Proof

This section proves that the protocol is a reliable multicast and tolerates f failures out of $f + 2$ processes. In fact the protocol tolerates any number of faulty processes but the problem is vacuous if there are less than two correct processes. In those situations, the protocol definition does not impose any particular behavior.

In Section 2.3 we exemplified the cases in which a process was failed. Here we formalize those cases as a set of conditions. A process is failed (or not correct) if:

- F1. The process does not follow the protocol or it crashed.
- F2. The process can not communicate with the TTCB or is impersonated by an attacker, e.g., if the attacker managed to capture the process' pair (eid, secret).
- F3. An attacker manages to falsify a MAC that should have been created by the process, e.g., if the attacker discovers one of the processes' symmetric keys.
- F4. The process can not send or receive successive copies of a message because its communication is systematically disrupted by an attacker.
- F5. The process does not get the result of an agreement because the TTCB discarded that result (the TTCB discards results after some time).

Theorem 1 *BRM-M is a reliable multicast protocol that tolerates f failed processes out of $n \geq f+2$ processes.*

Proof. The theorem is valid if the protocol verifies the three properties of Validity, Agreement and Integrity as defined previously (Section 3). The proof is developed in such a way that it imposes no limits on the number of faulty processes (only requires two correct processes). We prove each property in turn:

Validity. This property refers to a correct sender and a correct recipient. If a correct sender multicasts a message then a correct recipient in $group(M)$ (i.e., in $elist$) will eventually receive it, since F1, F2 and F4 are false. This will happen either due to the message sent in the first multicast (line 4) or to a retransmission (line 25). The correct recipient also receives the correct $H(M)$, since F1, F2 and F5 are false and the TTCB Agreement Service gives correct results. After receiving a message and the hash, the correct recipient will eventually deliver the message (F1 and F2 are false). This will happen either because phase 1 was completed successfully (line 9), or because it will eventually leave the loop (possibly after $Od + 1$ message multicasts) and end phase 2 (line 29).

Agreement. First, let us prove that if a correct recipient p delivers a message M then all correct recipients in $group(M)$ eventually receive M. If p is correct then it follows the protocol (F1 is false) and it delivers M only after: (1) receiving ACKs from all recipients in $group(M)$, i.e., in $elist$ (lines 8-9 and 22-23); or (2) sending $Od + 1$ copies of the message to every recipient from which it did not receive an ACK (line 24-26 and 28). If p receives an ACK from another recipient then, either the ACK was genuine (sent by the recipient) or fake. If the ACK was fake then the corresponding recipient is not correct (condition F3) and therefore the property of Agreement does not apply to this process. In case (2), if p sends $Od+1$ copies of a message to a recipient then either that recipient receives the message or it is failed and the property does not apply (F4). Therefore, if p delivers M then all correct recipients receive M.

Now we have to prove that if a correct recipient p' receives M then it eventually delivers M. If p' is correct it follows the protocol (condition F1) and it manages to communicate with the TTCB (condition F2). Since process p delivered M, then it must have obtained a correct $H(M)$ from the TTCB agreement. Therefore, since condition F5 is false, p' can also get $H(M)$ from the TTCB agreement. After receiving M and checking that it is the message corresponding to $H(M)$, p' will eventually deliver the message.

This proves that if a correct recipient delivers M then all correct recipients deliver M. A correct sender always delivers the message so this proves that the protocol verifies the Agreement property.

Integrity. For all messages with the same pair $(elist, tstart)$, every correct process runs a single instance of the protocol code. Additionally, an instance of the protocol always returns after delivering a message (lines 9 and 29). Therefore, every correct process delivers a message M at most once. Any correct process not in $group(M)$, i.e., not in $elist$, can not get $H(M)$ from the TTCB, therefore it can not deliver M. Now let us prove the second part of the property. The process $sender(M)$ is the process whose eid is the first in $elist$. The value returned by the agreement is the $H(M)$ proposed by the first element in $elist$ (decision TTCB_TBA_RMULTICAST in line 7), i.e., by $sender(M)$ since it is correct (F2). Therefore, the value of $H(M)$ returned by the agreement is always the value proposed by the sender. Consequently, a correct process can deliver a message M only if M was previously multicast by $sender(M)$, since a correct process follows the protocol (F1) and checks if the hash of the message it received is equal to $H(M)$ (assuming the hash function is collision resistant). □

6 Performance Evaluation

The experimental setting used to evaluate the protocol consisted of a COTS-based implementation of the TTCB described in [5]. The implementation of a TTCB has to be made highly secure to ensure (with high coverage) that our assumptions are not violated, otherwise, the protocol will not behave as expected. Therefore, the conceptual separation between local TTCBs and the operating systems has to be strongly enforced, since operating systems are in general attackable. This means, for instance, that local TTCBs could be implemented inside a hardware appliance board of some kind. In this case, the communication between that board and the rest of the host can be limited and therefore attacks against it can be prevented. The first implementation of the TTCB used a different approach. The local TTCB resides inside of a Real Time Linux Kernel (RT-Linux) and the separation is obtained by a set of security mechanisms.

This software-based version of the local TTCB has less coverage of the security assumptions than the one based on hardware, however, it is easier to deploy. Consequently, it becomes simpler to freely distribute the TTCB by the research community, allowing it to be tested and evaluated by other research groups. The TTCB control channel must also be protected. Currently, it is a dedicated Ethernet network, but other solutions are possible [5].

More specifically the performance results were obtained on a system with five PCs, each containing a Pentium III processor running at 450 Mhz and 64 Mbytes of main memory. The operating system of all PCs was RT-Linux. The PCs were connected by two 100 Mbps Fast-Ethernet LANs, one for the general purpose payload network and another for the internal control network of the TTCB. The protocol was implemented in C, compiled with the standard gcc compiler. The hash function that was utilized was MD5. Whenever possible, the communication among processes was based on IP multicast. Five processes were used in the tests, each one running on a distinct PC, and we assumed a setting where all processes were correct, i.e., no failed processes (*f = 0*). Throughout the experiments the value adopted for the omission degree was two (*Od = 2*). Each measurement was repeated at least 4500 times.

In the first set of experiments we tried to determine in which phase the protocol terminated. From the observed results, it is possible to conclude that for reasonable values of $tstart$, in the order of 2 ms, the protocol always terminates in the first (optimistic) phase. We noticed that although IP multicast is unreliable, all messages apparently reached the processes, and for this reason, they were able to propose their hash value before $tstart$ (see Figure 3). If messages were lost or if some of the processes were malicious, we would expect that in most cases the second phase would have had to be executed.

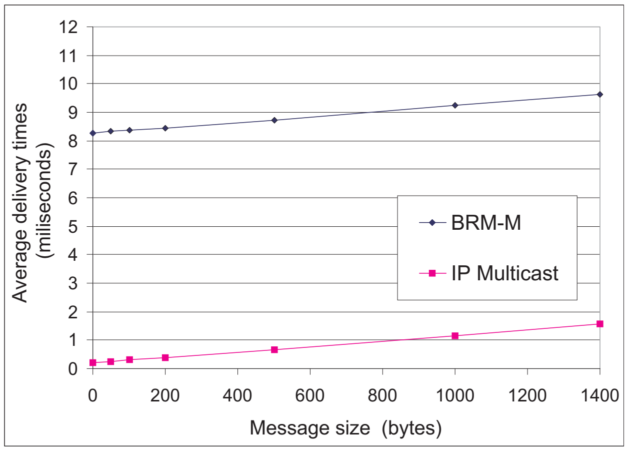

Figure 5. Five-node average delivery time for different message sizes.

In the second set of experiments we obtained message delivery times for the protocol. Since the protocol always finishes at the end of the first phase, it is possible to use the following methodology to calculate the delivery times. One of the processes is randomly selected as the sender, and then it reliably multicasts a message M of a given size. Then, immediately after delivery, a single recipient is selected to send a reply. This reply is an IP multicast for the same set of processes, with a message of the same size. For each execution of this procedure two times were measured: the round-trip time and the recipient processing time. The round-trip time (Trd) is obtained by the sender, and it corresponds to the time measured between the multicast and the reception of the reply. The recipient processing time ($Tproc$) is the time taken between the reception of the message M in the recipient and its reply. This time includes all tasks executed by the recipient, such as hash calculation, and it corresponds mostly to the time waiting for the TTCB Agreement Service, i.e., calling TTCB_propose and waiting for TTCB_decide to return the result of the agreement (lines 7-8). If one assumes that an IP multicast always takes the same amount of time, we can use the following formula to calculate the protocols message delivery time:

$$Td = (Trd - Tproc)/2 + Tproc \qquad (1)$$

Message size	Time (usec)	Standard dev
0	8273	2522
50	8335	2527
100	8378	2539
200	8426	2540
500	8727	2489
1000	9255	2524
1400	9634	2496

Table 1. Five-node average and standard deviation of the delivery time.

Figure 5 plots the average delivery time of the protocol (with 5 processes) as a function of the message data size. These results are compared with the unreliable IP Multicast (over UDP sockets) performance, also implemented in C and in the same environment of execution. In addition, Table 1 presents the delivery times together with the standard deviation for each package size.

The protocol overheads are mainly three: one IP multicast, some processing time (calculate the hash), and the execution of one TTCB Agreement. Figure 5 shows that the additional cost of the protocol in relation to an unreliable IP multicast is approximately 8 ms, on average. Since the processing time is in order of a few tens of microseconds, most of this cost corresponds to the waiting period due to the TTCB Agreement. Consequently, we expect our protocol will perform better as the TTCB is optimized, and faster protocols are used to implement the agreement service. Nevertheless, it should be noticed that the current performance results are already very good when compared with other Byzantine resilient protocols that have been published in the literature. For instance, in [18], for a group of 5 processes and message sizes of 0 and 1 Kbytes, the delivery times were approximately 47 and 50 ms, respectively.

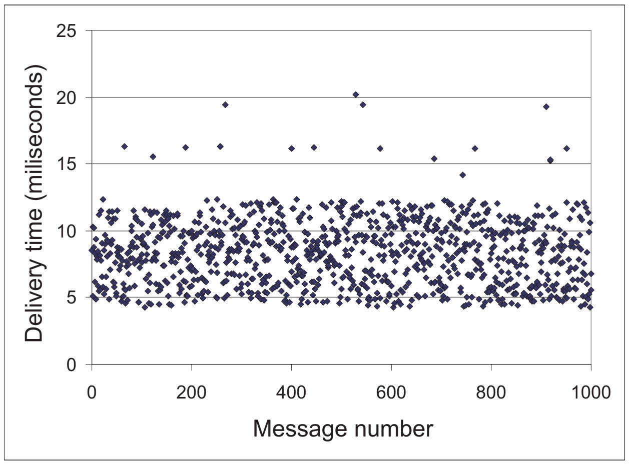

Figure 6. Five-node delivery times for 1000 messages with a size of 0 bytes.

The delivery time values exhibit a reasonably high standard deviation. Figure 6 displays the delivery times for 1000 executions of the protocol using a message data size of 0 bytes. The main explanation for this behavior is related to the internal implementation of the agreement service of the TTCB. Currently, it uses a time-triggered protocol where interactions with the network only happen every 4 ms (e.g., it only reads messages from the network at the beginning of the 4 ms interval). Therefore, an agreement will take more or less time depending on the instant when processes propose their values within the 4 ms interval.

7 Related Work

There is a significant amount of work in the area of reliable broadcasts for distributed systems – most of it, however, has focused on benign failures and/or assumed a synchronous model [7]. Reliable multicast protocols tolerating Byzantine faults make no assumptions about the behavior of faulty processes (similar to “Byzantine agreement” in the synchronous time model [9]). In asynchronous systems, it was proved a theoretical maximum that less than a third ($f \leq \frac{n-1}{3}$) process may be corrupted [2]. In our protocol, with the support of the TTCB, we can overcome this limit, and require only $f \leq n-2$.

The Rampart toolkit contains a reliable multicast protocol where processes communicate through authenticated reliable channels and use public-key cryptography to digitally sign some of the messages [18]. The protocol is based on a simple echo protocol where the sender starts by multicasting a hash of the message, then it expects a confirmation from a subset of the processes, and finally it multicasts the message (this protocol improves the echo protocol by Toueg [21] in terms of message complexity at the cost of more computation). Rampart assumes a dynamic membership provided by a protocol which also utilizes a three-phase commit strategy [19]. Later, Malki and Reiter optimized the Rampart protocol using a method of chaining acknowledgments to amortize the cost of computing the digital signatures through several messages [11]. Malkhi, Merrit and Rodeh proposed a secure reliable multicast protocol based on dissemination quorums, as a way to reduce delays specially in the case where $f \ll n$ [10]. This protocol assumes similar channels and uses public key signatures as the previous protocols, but considers, like in [11], static membership.

The SecureRing system provides a reliable message delivery protocol that uses public key cryptography and assumes a fully connected network [8]. The multicast is imposed on a logical ring, where a token controls who can send the messages. The Secure Trans protocol, which is implemented in the SecureGroup system, uses retransmissions and acknowledgments to achieve reliable delivery of messages [14]. These acknowledgments are piggybacked on messages that are themselves broadcasted. Each message is digitally signed to ensure authenticity and integrity. Both systems, SecureRing and SecureGroup, provide support for dynamic group membership changes.

There are some secure group communication systems which consider a non Byzantine failure model: Horus, Ensemble and Secure Spread. These systems assume that communication can be attacked but that hosts do not fail. Secure multicast protocols based on message authentication codes are given explicitly for Horus and Secure Spread [20, 1].

The BRM-M protocol does not need public key cryptography, one of the main bottlenecks of group communication performance [4], since it uses the TTCB to securely exchange a digest of the message. In terms of the network we have assumed unreliable channels, which results on a message complexity proportional to the omission degree. This paper does not discuss the membership service, but our protocol is suited for dynamic groups, such as those based on the MAFTIA architecture [16].

There is a body of research, starting with [13], on hybrid failure models that assume different failure type distributions for different nodes. For instance, some nodes are assumed to behave arbitrarily while others are assumed to fail only by crashing. Such a distribution might be hard to predict in the presence of malicious intelligent entities, unless their behavior is constrained in some manner. Our work might best be described as *architectural hybridization*, in the lines of works such as [15, 25], where failure assumptions are in fact enforced by the architecture and the construction of the system components, and thus well-founded.

8 Conclusion

The paper presents a new reliable multicast protocol for asynchronous systems with an hybrid failure model. This type of failure model allows some components to fail in a controlled way while others may fail arbitrarily. In our case, we assume the existence of a simple distributed security kernel, the TTCB, that can only fail by crashing, while the rest of the system can behave in a Byzantine way. By relying on the services of the TTCB, the protocol exhibits excellent behavior in terms of time and message complexity when compared with more traditional Byzantine protocols. Moreover, it only requires $n \geq f + 2$ correct processes, instead of the usual $n \geq 3f + 1$.

Besides describing a novel Byzantine-resilient protocol, the paper introduces the design of protocols based in our architectural hybrid failure model and, more specifically, the design of protocols using our distributed security kernel, the TTCB. In the future, we will pursuit these design principles to develop a complete suite of Byzantine-resilient group communication protocols, which is being produced in the context of project MAFTIA.

References

[1] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *Proc. the 20th IEEE International Conference on Distributed Computing Systems*, pages 330–343, Apr. 2000.

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

[3] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proc. the 19th ACM Symposium on Principles of Distributed Computing*, 2000.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. the Third Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[5] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proc. Fourth European Dependable Computing Conference*, Oct. 2002.

[6] F. Cristian and C. Fetzer. The timed asynchronous system model. In *Proc. the 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 140–149, 1998.

[7] 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.

[8] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proc. the 31st Annual Hawaii International Conference on System Sciences*, volume 3, pages 317–326, Jan. 1998.

[9] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[10] D. Malkhi, M. Merrit, and O. Rodeh. Secure reliable multicast protocols in a WAN. In *International Conference on Distributed Computing Systems*, pages 87–94, 1997.

[11] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. *The Journal of Computer Security*, 5:113–127, 1997.

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

[13] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proc. the 17th IEEE International Symposium on Fault-Tolerant Computing*, July 1987.

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

[15] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer-Verlag, Nov. 1991.

[16] D. Powell and R. J. Stroud, editors. *MAFTIA: Conceptual Model and Architecture. Project MAFTIA IST-1999-11583 deliverable D2*. Nov. 2001.

[17] M. O. Rabin. Randomized Byzantine Generals. In *Proc. the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[18] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proc. the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, Nov. 1994.

[19] M. K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, Jan. 1996.

[20] M. K. Reiter, K. P. Birman, and R. van Rennesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, Nov. 1994.

[21] S. Toueg. Randomized byzantine agreements. In *Proc. the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, Aug. 1984.

[22] P. Veríssimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bullettin of the Technical Committee on Operating Systems and Application Environments*, 7(4):35–39, 1995.

[23] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proc. the International Conference on Dependable Systems and Networks*, pages 533–542, June 2000.

[24] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. In *SIGCOMM*, pages 83–93, 1989.

[25] P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global clock service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, 1997.