A Simple Intrusion-Tolerant Reliable Multicast Protocol using the TTCB *

Lau Cheuk Lung[1] **, Miguel Correia**[2] **, Nuno Ferreira Neves**[2] **, Paulo Veríssimo**[2]

[1]PPGIA - Programa de Pós-Graduação em Informática Aplicada
PUC-PR - Pontifícia Universidade Católica do Paraná
R. Imaculada Conceição, 1155 - Prado velho - CEP 80215-901 - Curitiba -PR
lau@ppgia.pucpr.br

[2]Faculdade de Ciências da Universidade de Lisboa
Bloco C5, Campo Grande
1749-016 Lisboa - Portugal
{mpc,nuno,pjv}@di.fc.ul.pt

Abstract. *This paper proposes a simple reliable multicast protocol that tolerates arbitrary faults, including malicious faults such as intrusions. The goal is to show a novel way of designing intrusion-tolerant protocols based on a well-founded hybrid fault model. This model is based on a simple distributed security kernel – the TTCB – which is used by the processes only to execute securely critical steps of the protocol. Otherwise, the processes and their communication can be attacked in unlimited ways. The TTCB provides only a few basic services, which allow our protocol to tolerate a number of faults similar to accidental fault-tolerant protocols: for f faults, our protocol requires $f + 2$ processes, instead of $3f + 1$ in typical intrusion-tolerant (or Byzantine) protocols. The protocol exhibits fast termination in the presence of intrusions and/or crash or malicious process failures, since it does not use any cryptography in runtime.*

Resumo. *Este artigo propõe um protocolo de difusão confiável simples que tolera faltas arbitrárias, incluindo faltas maliciosas tais como intrusões. O objetivo é mostrar um modo original de projetar protocolos tolerantes a intrusões baseados num bem fundamentado modelo de faltas híbrido. Este modelo é baseado num kernel de segurança distribuído simples – o TTCB – o qual é usado pelos processos somente para executar passos críticos do protocolo de forma segura. O TTCB fornece somente alguns serviços básicos, os quais permitem ao nosso protocolo tolerar um número de faltas similar aos protocolos tolerantes a faltas acidentais: para f faltas, nosso protocolo requer $f + 2$ processos, ao invés de $3f + 1$ nos protocolos tolerantes a intrusões (ou Byzantinos) típicos. O protocolo exibe terminação rápida na presença de intrusões e/ou crash ou processos maliciosos, uma vez que não usa qualquer criptografia em tempo de execução.*

*This work was partially supported by the EC, through project IST-1999-11583 (MAFTIA), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LASIGE) and projects POSI/1999/CHS/33996 (DEFEATS) and POSI/CHS/39815/2001 (COPE).

1. Introduction

Distributed protocols that tolerate arbitrary faults – also called *Byzantine faults* – have been studied for some time now (see, e.g., [Lamport et al., 1982, Rabin, 1983, Fischer, 1983]). Recently, with the Internet going mainstream and the rising tide of malicious activity, a new interest for these protocols emerged under the designation of *intrusion tolerance*. These protocols usually consider a set of cooperating processes (or hosts) 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 asynchronous time model is usually used by this type of protocols because it describes appropriately the Internet and other networks with unpredictable timeliness, although more or less subtle synchrony assumptions usually have to be done. Examples of recent intrusion-tolerant protocols can be found in [Castro and Liskov, 1999, Reiter, 1994, Malkhi et al., 1997, Kihlstrom et al., 1998, Moser et al., 2000, Cachin et al., 2000].

Intrusion-tolerant protocols based on the asynchronous model normally have to assume a maximum number of processes that are allowed to fail. For the particular problem addressed in this paper – reliable multicast – Bracha and Toueg showed that it is impossible to send reliable multicast if there are more than $f = \frac{n-1}{3}$ faulty processes in a system with n processes [Bracha and Toueg, 1985]. Usually these protocols are also slow when compared to accidental fault-tolerant protocols because they require several rounds of message exchange and asymmetric cryptography to protect the transmitted messages (see, e.g., [Reiter, 1994]).

This paper presents a reliable multicast protocol based on a hybrid fault model. The basic idea of this type of models 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 fail in an arbitrary way, however, we assume the existence of a distributed security kernel which can only fail by crashing. This kernel is called the Trusted Timely Computing Base (TTCB). It has a set of features that together are innovative: it is distributed with its own private control channel, it is secure and real-time, and it provides a limited set of services. The design and implementation of the TTCB were presented and discussed in another paper [Correia et al., 2002b].

The intrusion-tolerant reliable multicast protocol presented here – BRM-T – is based on the idea of using the TTCB to give the processes a reliable digest of the sent message. The protocol terminates early even if there are faulty processes or the network behaves badly. The fact that it does not need any cryptography in runtime and its simplicity are the two main differences in relation to a protocol of the same family presented elsewhere [Correia et al., 2002a]. BRM-T is also efficient in relation to other protocols in the literature, precisely because it does not use cryptography. Also, on the contrary to similar protocols, BRM-T does not impose any limit on the maximum number of faulty processes. This result is similar to what is achieved in *synchronous* systems and is usually stated as $n \geq f + 2$, since the problem is vacuous if there are less than two correct processes.

2. The System Model and the TTCB

The TTCB is a *secure* and *real-time* distributed subsystem with local parts in the hosts and a control channel. The architecture of a system with a TTCB is presented in Figure 1. Each host contains the typical software layers: the operating system and runtime environments, the applications and processes, etc. The local parts, or *local TTCBs*, are computational components with activity, conceptually separate from the 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. We will not take more time discussing how these conceptual separations are obtained in practice, since the implementation of the TTCB was already presented [Correia et al., 2002b].

The system's time model is mostly asynchronous, with the exception of the TTCB, which is real-time, therefore synchronous. In relation to the (asynchronous) payload system, 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.

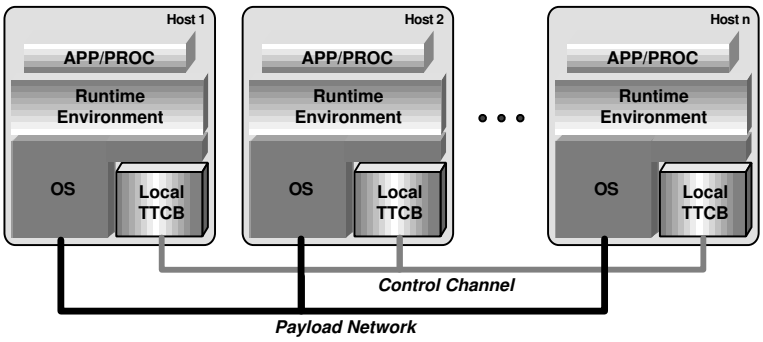

Figura 1: System with a TTCB.

The protocol presented in this paper uses only three TTCB services:

- *Local authentication.* This service allows a process to communicate securely with the local TTCB. The service authenticates the local TTCB before the process, and establishes a shared symmetric key between both[1]. This shared symmetric key is used to implement a *secure channel* between the process and the local TTCB, and then their exchanges can be authenticated, encrypted, etc. depending on what is needed [Correia et al., 2002b].
- *Trusted block agreement.* This 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.
- *Trusted Absolute Timestamping Service.* This service provides globally meaningful timestamps. It is possible to obtain timestamps with this characteristic because

[1]Every local TTCB has an asymmetric key pair. We assume that a process is able to get a trustworthy copy of the public key. This key is then used to authenticate the local TTCB.

local TTCBs clocks are synchronized to a precision π, i.e., for every pair of local TTCBs (i, j): $|C_i(t) - C_j(t)| < \pi$.

Next we describe with more detail the trusted block agreement service which is crucial for the understanding of the protocol.

2.1. Trusted Block Agreement Service

The *agreement service* (for short) is 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.

The interface of the agreement service has two functions:

```
outp ←TTCB_propose(eid, elist, tstart, decision, value)

outd ←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 $outp$ with two fields: *outp.error* is an error code and *outp.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. *outd* is a record with four fields: *outd.error* gives an error code; *outd.value* is the value decided; *outd.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; *outd.proposed-any* is a similar mask but that indicates which processes proposed any value.

2.2. Process Failure Modes

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:

1. The process crashes or starts to behave maliciously
2. The pair $(eid, secret)$, which lets the process communicate securely with the local TTCB, is discovered by a local attacker

3. The process is unable to exchange data with other processes because its communication is systematically disrupted or delayed

The first case is the most obvious. We consider that a malicious process can fail arbitrarily (e.g., in a Byzantine way). 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.

Case two describes a situation that might lead to a personification attack. 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. If this pair is kept secret, the attacker can only try to disrupt or delay the communication between the process and the local TCCB – personification attacks are prevented.

The third case requires some discussion. Asynchronous protocols typically assume 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, let us say, 10 hours 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 [Veríssimo et al., 1989]. 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 that will be used in the protocols. If Od is set to a very high value, then our protocols 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 [Chandra and Toueg, 1996]. Failure detection mechanisms are out of the scope of this paper.

Another advantage of considering systematically delayed processes as failed is related with the implementation of the TTCB. Since the TTCB is a small component, it can only keep the results of the agreement service for a limited time. If a delayed process asks for a result after it expired the simplest thing to do is to consider the process as failed. Alternatively, the protocols could be made more complex to recover from this situation. However, there is no much justification in doing so for the reason pointed earlier – if the process is too late it is useless.

3. The Reliable Multicast Protocol

In all multicast protocols processes play one of two roles: sender or recipient. A message transmitted to a group of processes should be delivered to all members, including the sender. This is not always possible due to failures. 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: 1) all correct processes deliver the same messages, and 2) if a correct sender transmits a message then all correct processes deliver this message [Bracha and Toueg, 1985]. These rules do not imply that the message will be delivered in the 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. Reliable multicast also gives no assurances about the order in which messages are delivered. Each process can deliver its messages in a distinct order.

Formally, a reliable multicast protocol has the following properties [Hadzilacos and Toueg, 1994]:

- *Validity:* If a correct process multicasts a message M then a correct process in *group(M)* [2] 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)* [3] is correct then M was previously multicast by *sender(M)*.

3.1. The BRM-T Protocol

An execution of the BRM-T protocol consists in the sender multicasting the message to all recipients, and then each recipient multicasting again the message to all others. Each multicast is performed $Od + 1$ times in order to tolerate omissions due to accidental faults (see Section 2.2.). The recipients multicast is needed to ensure the Agreement property, in case a malicious sender only transmits the message to a subset of the correct processes. The authenticity and integrity of the message is checked using an hash code sent through the TTCB agreement service. Consequently, messages do not have to carry any cryptographic signatures and processes do not need to share any cryptographic keys.

[2] Sender and recipients of M
[3] Sender of M

BRM-T Sender protocol

1 *tstart = TTCB_getTimestamp() + T_0;*
2 *M := (elist, tstart, data);*
3 *propose := TTCB_propose(elist, tstart, TTCB_TBA_RMULTICAST, H(M));*
4 **repeat** *Od+1 times* **do** *multicast M to elist except sender* **od**
5 *deliver M;*

BRM-T Recipient protocol

6 *read_blocking(M);*
7 *propose := TTCB_propose(M.elist, M.tstart, TTCB_TBA_RMULTICAST, $\perp$);*
8 **do** *decide := TTCB_decide(propose.tag);*
9 **while** *(decide.error $\neq$ TTCB_TBA_ENDED);*
10 **while** *(H(M) $\neq$ decide.value)* **do** *read_blocking(M)* **od**
11 **repeat** *Od+1 times* **do** *multicast M to elist except sender* **od**
12 *deliver M;*

Figura 2: BRM-T protocol.

Figure 2 shows an implementation of the protocol. A message consists of a tuple with three components $(elist, tstart, data)$. $elist$ is a list of $eids$ with the format accepted by the TTCB agreement service. The first element of the list is the eid of the sender, and the others are the $eids$ of the receivers. $tstart$ is the timestamp that will be given to the agreement service, and $data$ is the information to be transmitted. Each execution of the protocol is identified by $(elist, tstart)$. The $read\_blocking()$ primitive only reads messages with the value $(elist, tstart)$ corresponding to the protocol instance being executed. Other values of the pair are processed by other instances of the protocol. We assume that there is a garbage collector that discards messages for instances of the protocol that have already finished running. This garbage collector can be constructed by keeping in a list with the identifiers of the messages already delivered and comparing the identifiers of the arriving messages.

The *sender protocol* receives as arguments *elist* and *data* from the application and starts by calculating *tstart* and constructing the message (lines 1-2). Constant T_0 has to be carefully determined because, on one hand, smaller values ensure faster termination of the protocol. On the other hand, T_0 should be sufficiently large for *TTCB_propose* to be called before $tstart$ (otherwise, the value would not be accepted). Since the system is asynchronous, it is not possible to establish the best value for T_0 because delays are nondeterministic, but, in practice, a high probable value can be defined. Nevertheless, if *TTCB_propose* is called too late, the error $TTCB\_TSTART\_EXPIRED$ is returned and the sender protocol can either retry with a new $tstart$ or return an error (for simplicity this condition does not appear in the code). The agreement execution is defined by *elist*, *tstart* and the decision function (line 3). Both sender and recipients use the same decision function $TTCB\_TBA\_RMULTICAST$. This function selects as the result of the agreement the value proposed by the first process in $elist$, which in our case is the sender. The value proposed by the sender is the hash of the message, $H(M)$. An hash function is basically a one-way function that compresses its input and produces a fixed sized digest (e.g., 128 bits for MD5). In the paper, we assume that an attacker is unable to subvert the properties of the hash function, including weak and strong collision resis-

tance [Menezes et al., 1997]. The sender terminates by multicasting the message $Od + 1$ times and then delivering it (lines 4-5).

BRM-T uses the agreement service in the way described to multicast $H(M)$ to the recipients, and the fact that the TTCB is involved in the procedure guarantees that all recipients receive a reliable H(M). Given the collision resistance property of the hash function, it is "computationally infeasible" for a malicious sender to give to two correct recipients distinct messages but with the same $H(M)$. The fact that an instance of the protocol is defined by the pair $(elist, tstart)$, that together with the decision function uniquely identifies the agreement, is also very important. This prevents an attacker or malicious process from trying to change $elist$, for instance, to exclude a process, or from modifying $tstart$, for instance, to delay the protocol. If one process gives different values for these parameters to the agreement service, a new instance of agreement will be created in the TTCB, and its execution will not interfere with the original execution of the agreement.

The *recipient protocol* starts by waiting to read a message (line 6). Then, it calls *TTCB_propose* with the sole objective of getting the tag that identifies the agreement (line 7). Next, there is a call to *TTCB_decide*, which is done inside a loop to guarantee that the protocol waits for the agreement to finish, in case it is still running (lines 8-9). If the received message is different from the one transmitted by the sender then $H(M)$ will be different from the value returned by the agreement, $decide.value$. In this case, the protocol has to wait for the correct message to be received (line 10). To complete the protocol, the process multicasts the message $Od + 1$ times to the other recipients and delivers it (lines 11-12).

As mentioned in Section 3., there are situations when the protocol does not terminate if the sender is malicious or the process is failed. First, in some cases, the recipient may never receive the message (line 6 and 10): (1) if the sender is malicious it may not send the message to any correct recipient (e.g., multicasts $M1$ and gives to the agreement $H(M)$); or (2) if all the messages sent to the process are corrupted or lost, which means, according to our process model, that the recipient is faulty. Second, a malicious sender might send a message but never propose a value. In this case, the agreement at the correct recipients will not terminate and they will be blocked forever (lines 8-9). Third, if a recipient becomes aware that the protocol is running so late that when it calls $TTCB\_decide$ the result of the agreement is no longer available. In this case the recipient is considered faulty, as mentioned in Section 2.2..

Example Execution of the BRM-T Protocol

Figure 3 illustrates the behavior of the BRM-T protocol. The horizontal lines represent the execution of processes through time. The thicker line represents the TTCB as a whole, even though, each process calls a separate local TTCB in its host (this representation is used for simplicity).

The sender calls the TTCB agreement and then multicasts the message twice ($Od = 1$). These messages are received in the following way: P2 receives the two copies of the message, P3 receives the first copy corrupted and the second well, and P4 does not receive the first copy and the second is delayed. The example assumes that the first message sent to P3 is corrupted only in the $data$ part, and for that reason it is still possible

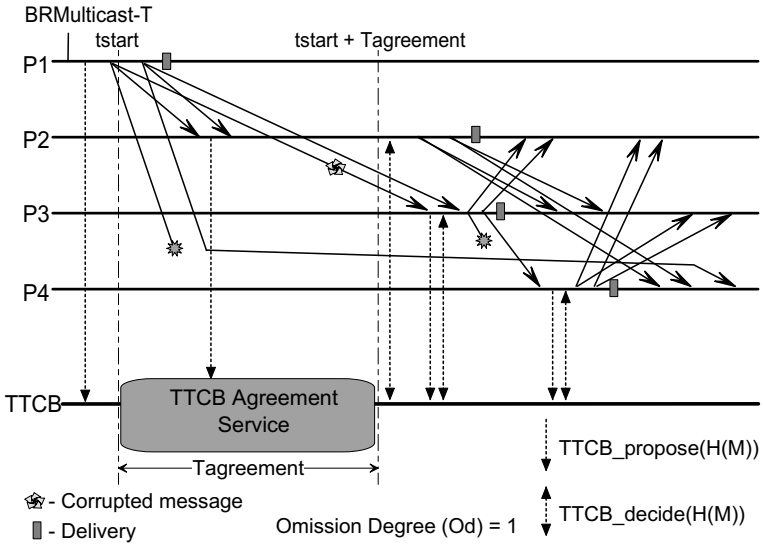

Figura 3: Example execution of the BRM-T protocol.

to determine this protocol instance. When a message arrives, the recipient calls the TTCB agreement to get the result with the reliable value of $H(M)$. Both processes P2 and P3 get this value almost immediately after the end of the agreement. They use the hash to select which of the messages they received is correct, and then they multicast the message to all the other recipients. P4 asks for the result of the agreement later, when it receives the first message from the protocol. Then, it multicasts the message.

4. Protocol Evaluation

This section evaluates the BRM-T protocol in terms of message complexity and time performance.

Message Complexity

The number of messages transmitted by the BRM-T protocol is always the same. However, the way these messages are counted depends on how the unreliable multicast primitive is implemented. Equations are given for implementations in which a multicast is performed using a network multicast primitive (e.g., IP multicast) or with consecutive calls to a point-to-point send primitive (e.g., UDP).

If there are f failed processes that do not communicate (e.g., because they crashed), then the number of messages send by the $(n - f)$ correct processes is:

$$N_{mcast} = (Od + 1)(n - f) \quad (1)$$

$$N_{p2p} = (Od + 1) \times ((n - 1) + (n - f - 1)(n - 2)) \quad (2)$$

Time Performance

The performance of the BRM-T protocol was evaluated using the current implementation of the TTCB, the *COTS-based TTCB* [Correia et al., 2002b]. This setup is based on six

450 Mhz Pentium III PCs with 192 Mbytes RAM. The payload network and the TTCB control network are 100 Mbps Fast-Ethernet LANs. Each PC has two network adapters. The operating system is RTAI [4], a freeware real-time engineering of Linux. RTAI was security-enhanced in order to prevent intrusions in the local TTCB and in the TTCB control network. The code was implemented in C and compiled using the standard *gcc* compiler. The hash function used was MD5 [Menezes et al., 1997] and the communication was done with IP multicast. $T_{agreement}$ is currently $13ms$.

A first set of experiments was used to setup some protocol parameters. Setting up $T_0 = 1ms$ the sender managed always to propose before *tstart* (lines 1-3). We also observed that although IP multicast is unreliable, if there are no attacks usually no messages are lost.

All other experiments measured the time the protocol takes to deliver the message to a recipient. There was always one sender and five recipients, one per PC. The methodology used was the following. One of the processes is randomly chosen as the sender and multicasts a message M using an IP multicast address A. When the recipients receive the message they follow the protocol and resend the message to all other recipients using an IP multicast address B (the sender does not listen to this address). Then, immediately after delivery, a recipient is selected to answer the sender. This reply is an IP multicast for the same set of processes using the address A, and with a message of the same size. For each execution of the protocol 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 last 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 8-9). If we assume that an IP multicast always takes the same amount of time, we can use the following formula to calculate the protocol's message delivery time:

$$Td = \frac{Trd - Tproc}{2} + Tproc \quad (3)$$

Each measurement is the average of at least 1000 executions of the protocol. Both the network and the PCs were lightly loaded.

Figure 4 shows the variation of the BRM-T average delivery times with the message data size, and the standard deviation (second set of experiments), with no faults. The messages carry an extra 24 bytes header for BRM-T. The picture compares these times with the unreliable IP multicast (under UDP sockets) delivery times with the same message sizes. The gap of approximately $8ms$ between both is due mostly to the TTCB Agreement execution time. We are currently working on an faster protocol for this TTCB service, that will have a direct impact on the performance of BRM-T. Nevertheless, the times obtained seam to be good when compared with protocols in the literature. For instance, in [Reiter, 1994], for 6 processes and messages with 0 and 1 Kbytes, the de-

[4] http://www.rtai.org

livery times were approximately 52 and 57 ms, about 6 times the BRM-T times. This comparison should, however, be taken with caution since the test environment were quite different.

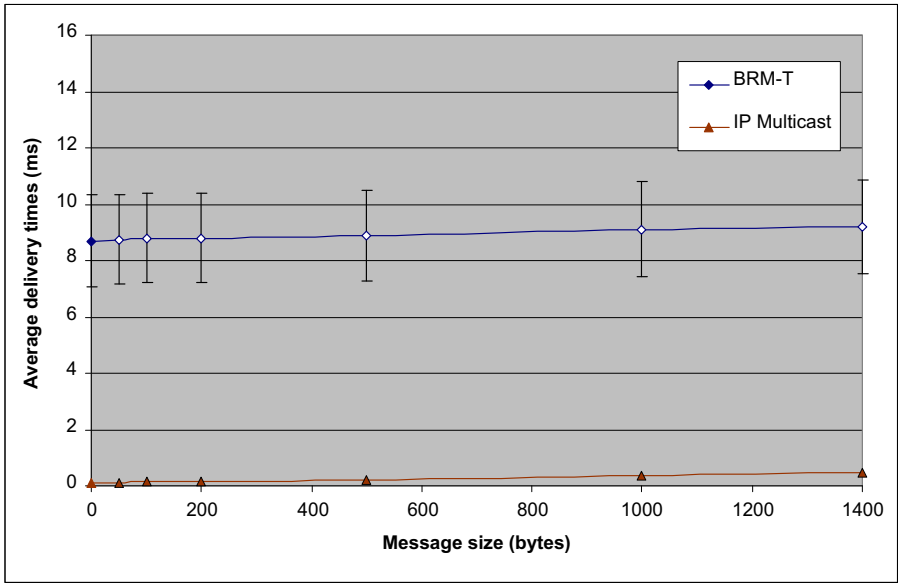

Figura 4: Six-node average delivery time for different message sizes, with $Od = 2$.

The third set of experiments assessed the impact of the omission degree on the protocol delivery times. Figure 5 shows the variation of the average delivery time with Od and the respective standard deviations. Varying Od from 0 (1 copy sent) to 10 (11 copies sent) the performance was only slightly affected.

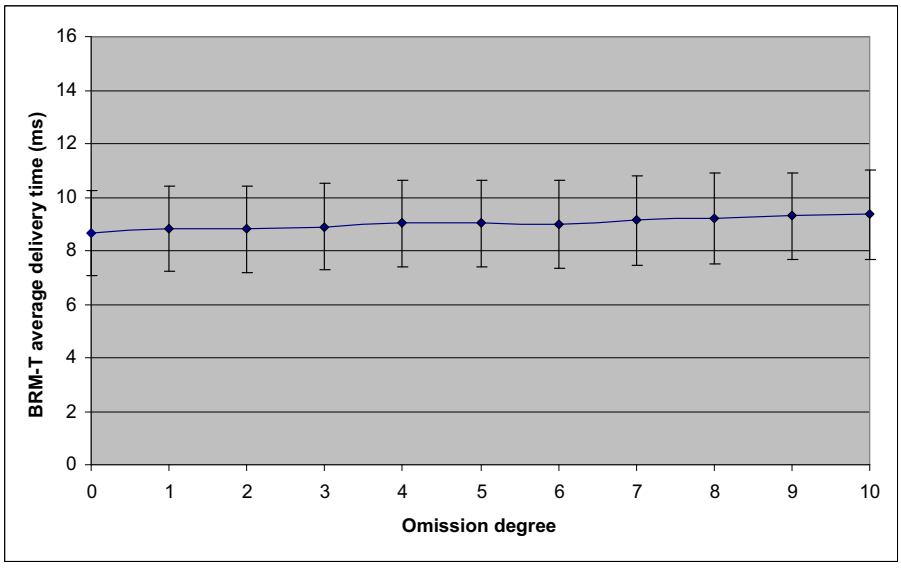

Figura 5: Six-node BRM-T average delivery time for different Od values (data size $500bytes$).

The fourth set of experiments assessed the impact of silent processes in the performance of the protocol. A process can be silent for a number of reasons, for instance, because it crashed or because it is malicious and does not want to execute the protocol.

Figure 6 shows the average delivery times obtained with 0 to 4 silent processes. The difference between the delivery times with none and one or more silent processes is around $4ms$. Why does the protocol takes longer to execute when there are silent processes? Because in all previous experiments processes usually proposed before $tstart$ and therefore the TTCB Agreement started to run earlier and also terminated sooner. When there are silent processes, the Agreement starts running only by $tstart$ and is considered to terminate only by $tstart + Tagreement$, which is a pessimistic value because $Tagreement$ is the higher time the Agreement may possible take to run.

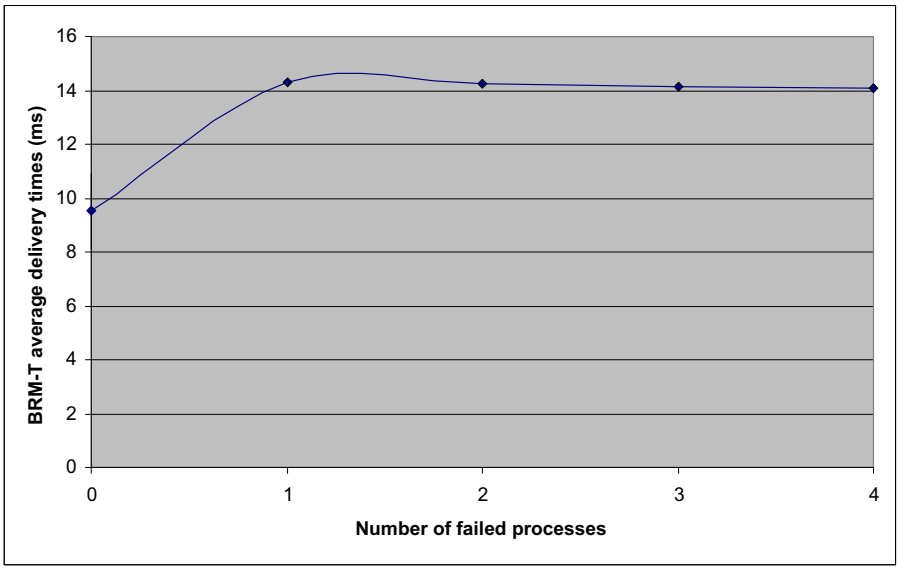

Figura 6: Six-node average delivery time with 0 to 4 silent processes ($Od = 2$, data size $500bytes$).

5. Related Work

Early reliable multicast protocols which tolerate arbitrary or "Byzantine" faults were reported both considering the synchronous [Lamport et al., 1982] and the asynchronous time model [Bracha and Toueg, 1985]. Bracha and Toueg established the result that in asynchronous systems less than a third ($f < n/3$) of the processes can be corrupted for the protocol properties to hold [Bracha and Toueg, 1985]. In our protocols, with the support of the TTCB, we can overcome this limit, and require only $f \leq n - 2$, a result analogous to the one obtained in synchronous systems [Lamport et al., 1982].

The Rampart toolkit contains a reliable multicast protocol that uses public-key cryptography to digitally sign some of the messages [Reiter, 1994]. The protocol is based on a simpler echo protocol that improves an earlier echo protocol by Toueg [Toueg, 1984]. Rampart assumes a membership service [Reiter, 1996]. Later, Malki and Reiter optimized the Rampart protocol using a method of chaining acknowledgments [Malkhi and Reiter, 1997]. 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 $f \ll n$ [Malkhi et al., 1997]. These two protocols have a static membership.

The SecureRing system provides a reliable message delivery protocol that uses public key cryptography and assumes a fully connected network [Kihlstrom et al., 1998]. The multicast is imposed on a logical ring, where a token controls who can send the messages. The Secure Trans protocol, part of the SecureGroup system, uses retransmissions and acknowledgments to achieve reliable delivery of messages [Moser et al., 2000]. Both systems support dynamic group membership.

The BRM-T protocol does not need public key cryptography (asymmetric cryptography), one of the main bottlenecks of group communication performance [Castro and Liskov, 1999], since it uses the TTCB to securely exchange a digest of the message. In fact, it also does not even use symmetric cryptography in runtime, one of the main differences in relation to a protocol of the same family called BRM-M [Correia et al., 2002a]. BRM-T and BRM-M have some points in common and some differences. The first, BRM-T, is based on the well known idea of multicasting a message enough times to tolerate accidental omissions in the network [Veríssimo et al., 1989]. The protocol terminates early even if there are faulty processes (accidental or malicious faults) or the network behaves badly, and it does not need to use cryptography in the messages, therefore avoiding the maintenance of shared keys among processes. This protocol, however, is pessimistic in the sense that it potentially sends more messages than what is necessary. The second protocol, BRM-M, uses acknowledgements. Consequently, it usually sends fewer messages but may take longer to terminate if there are faulty processes. This protocol requires message authentication codes (MAC) only in the acknowledgements (not in the data messages), but since these are based on symmetric cryptography the performance does not suffer too much [Castro and Liskov, 1999, Menezes et al., 1997]. We argue that providing two similar protocols optimized for different conditions can be useful for building systems which adapt to, e.g., different levels of threat. In terms of the network both protocols assume unreliable channels, which results on a message complexity proportional to the omission degree.

There is a body of research on *hybrid fault models* starting with the work by Meyer and Pradhan [Meyer and Pradhan, 1987] which assumes different failure type distributions for different nodes. For instance, a number of nodes is allowed to fail arbitrarily while others can fail only by crashing. These distributions are hard to substantiate in the presence of malicious and intelligent entities, unless their behavior is constrained in some way. Our model may be better called an *architectural hybrid fault model* in the sense that it makes different failure mode assumptions about different system components, and that those assumptions are enforced by construction.

6. Conclusion

This paper presents a new intrusion-tolerant reliable multicast protocol for asynchronous systems with an hybrid fault 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, both protocols exhibit excellent behavior in terms of time and message complexity when compared with more traditional Byzantine protocols. Moreover, they only require $n \geq f + 2$ correct processes, instead of the usual $n \geq 3f + 1$.

Referências

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

Cachin, C., Kursawe, K., and Shoup, V. (2000). Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132.

Castro, M. and Liskov, B. (1999). Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186.

Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

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

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

Fischer, M. J. (1983). The consensus problem in unreliable distributed systems (A brief survey). In Karpinsky, M., editor, *Foundations of Computing Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer-Verlag.

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

Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. (1998). The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, pages 317–326.

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

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

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

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

Meyer, F. and Pradhan, D. (1987). Consensus with dual failure modes. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, pages 214–222.

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

Rabin, M. O. (1983). Randomized Byzantine Generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409.

Reiter, M. (1994). Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80.

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

Toueg, S. (1984). Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178.

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