Worm-IT – A Wormhole-based Intrusion-Tolerant Group Communication System

Miguel Correia [a], Nuno Ferreira Neves [a], Lau Cheuk Lung [b], Paulo Veríssimo [a]

[a] *Faculdade de Ciências da Universidade de Lisboa, Campo Grande, Bloco C6, Piso 3, 1749-016 Lisboa, Portugal*

[b] *Programa de Pós-Graduação em Informática Aplicada, Pontifícia Universidade Católica do Paraná, Rua Imaculada Conceição, 1155, 80.215-901, Brasil*

Abstract

This paper [1] presents Worm-IT, a new intrusion-tolerant group communication system with a membership service and a view-synchronous atomic multicast primitive. The system is intrusion-tolerant in the sense that it behaves correctly even if some nodes are corrupted and become malicious. It is based on a novel approach that enhances the environment with a special secure distributed component used by the protocols to execute securely a few crucial operations. Using this approach, we manage to bring together two important features: Worm-IT tolerates the maximum number of malicious members possible; it does not have to detect the failure of primary-members, a problem in previous intrusion-tolerant group communication systems.

Key words: Byzantine fault tolerance, intrusion tolerance, group communication, view synchrony, asynchronous distributed algorithms

1 Introduction

Group communication is a well-known paradigm for the construction of distributed applications. This paradigm has been successfully used to support a large range of

[1] Appeared as: Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, Paulo Verssimo. Worm-IT - A Wormhole-based Intrusion-Tolerant Group Communication System. Journal of Systems & Software, vol. 80, n. 2, pages 178-197, Elsevier, February 2007

Email addresses: nuno@di.fc.ul.pt (Nuno Ferreira Neves), lau@ppgia.pucpr.br (Lau Cheuk Lung), pjv@di.fc.ul.pt (Paulo Veríssimo).

fault-tolerant applications, from databases to web servers. Some examples of current applications are the Internet Seismic Processing System (INSP) [2], the Zope Replication Service [3] and PostgreSQL-R [4]. These applications use group communication to support replication, thus increasing fault tolerance.

The two main components of a group communication system are the membership and the communication services. The *membership service* is the component in charge of keeping an updated list of the group members, processing joins and leaves of the group, and assessing the failure of members. The *communication service* provides primitives for data transmission in the group, e.g., reliable, causal order or total order multicasts.

This paper presents the design and evaluation of the *Wormhole-based Intrusion-Tolerant Group Communication System (Worm-IT)*. This system appears in the context of recent work in *intrusion tolerance*, i.e., on the application of fault tolerance concepts and techniques to the security field (Fraga and Powell, 1985; Adelsbach et al., 2002; Veríssimo et al., 2003). A system is intrusion-tolerant if it tolerates arbitrary faults, including both accidental and malicious faults, such as attacks and intrusions (also called *Byzantine faults* in the literature after Lamport et al. (1982)). In other words, the system should continue to provide correct services and follow its specification despite a number of intrusions in the processors and attacks in the network (e.g., delay, modification, or replay of messages), so that it can be used to implement secure distributed applications. For instance, take the example applications above that use group communication to tolerate accidental faults. These applications run in the Internet so the replicas can be attacked by hackers, viruses or worms. Worm-IT would allow these applications to run as expected if even a certain number of replicas were successfully attacked.

Most work in group communication has considered only crash failures (see Chockler et al. (2001) for a survey on this topic). Some of these earlier systems evolved for a weaker model, which considers that the communication can be attacked, e.g., Ensemble (Rodeh et al., 2001) and Secure Spread (Amir et al., 2005). More recently, interest emerged in designing group communication systems for environments that may suffer arbitrary faults, including attacks and intrusions: Rampart (Reiter, 1994, 1996), SecureRing (Kihlstrom et al., 2001) and SecureGroup (Moser et al., 2000; Moser and Melliar-Smith, 1999). Project ITUA implemented an enhanced version of Rampart (Ramasamy et al., 2002). These systems guarantee not only the security of their communication, but also that their specification is attained even if some of the nodes are attacked and start to behave maliciously. All of them, with the exception of SecureGroup, have optimal resilience, i.e., they tolerate the failure of at most $f = \lfloor \frac{n-1}{3} \rfloor$ out of n members.

[2] http://www.3dgeo.com/products/insp.html
[3] http://www.zope.com/products/zope_replication_services.html
[4] http://gborg.postgresql.org/project/pgreplication/

Worm-IT is composed of a membership service and a view-synchronous atomic multicast primitive (VSAM). It is based on the primary partition model, i.e., only processors in the the partition with more processors remain in the group if the network is partitioned for some reason (Birman, 1997).

Contributions

Every one of these past intrusion-tolerant group communication systems assumes a homogeneous environment: any component can be equally attacked and fail (although only f members can fail in a window of time); and both the processing and message delivery delays are unknown (i.e., the system is asynchronous). Albeit this model looks simple, all systems with optimal resilience – both Rampart/ITUA and SecureRing – suffer from a theoretical problem with practical consequences. They depend on the correct behavior of a special group component, the *primary-member* (called coordinator in Rampart/ITUA, and token-holder in SecureRing). If this member crashes or suffers an intrusion and starts to behave maliciously, the system might be prevented from making progress (e.g., from continuing to deliver messages). Therefore, the liveness of the system depends on the detection and removal of a faulty primary, something that usually requires assumptions about the maximum execution times of certain primary-member operations (e.g., maximum period T_{max} to respond to a specific request). This is a theoretical problem because in asynchronous systems one cannot (or should not) make any time assumptions. Moreover, from a practical point of view, this also has consequences, namely on the performance and security of the system. For instance, assuming a long T_{max} period will cause the system to take a long time to recover from a crash in the primary. On the other hand, assuming a shorter period will make the system vulnerable to attacks that delay the primary, originating a false detection and its eviction from the group. This allows malicious attackers to impair the assumption that no more than $f = \lfloor \frac{n-1}{3} \rfloor$ out of n members fail, by causing the removal of correct (primary) members from the group.

This paper takes a different approach that does not suffer from this problem. Although the difficulty of building complex systems that are secure is conspicuous, it is currently feasible to build secure distributed systems with limited functionality. In this paper, we consider that most of the system has the same characteristics as above: insecure and with uncertain timeliness. However, we also assume the existence of a special secure distributed component called *Trusted Timely Computing Base (TTCB)* (Correia et al., 2002b). The purpose of this component is to provide a small number of simple services to protocols or applications. Worm-IT runs to most extent in the "normal" system, but occasionally uses the services provided by the TTCB to execute some critical steps of its protocols. In this innovative kind of system architecture, these privileged components have been called *wormholes* (Veríssimo, 2003). Therefore, Worm-IT can be constructed in a completely decen-

tralized way, avoiding the above-mentioned problem, at the cost of depending on a distributed secure component – the TTCB.

The main contributions of the paper are: (1) the presentation and evaluation of a modular intrusion-tolerant group communication system with optimal resilience that does not suffer from the problem of detecting the failure of the primary discussed above; and (2) the description of the first reasonably complex system based on our novel wormhole-enhanced architecture.

2 The TTCB

Like any other distributed system, Worm-IT uses several support services. Examples are operating system calls, software libraries and communication primitives. Worm-IT uses also a set of services provided by a distributed component: the TTCB (Correia et al., 2002b). We present the TTCB component right away, even before the system model, due to its novelty and importance to the presentation of the system. Moreover, the system model depends on the TTCB, so it is convenient to presented it first.

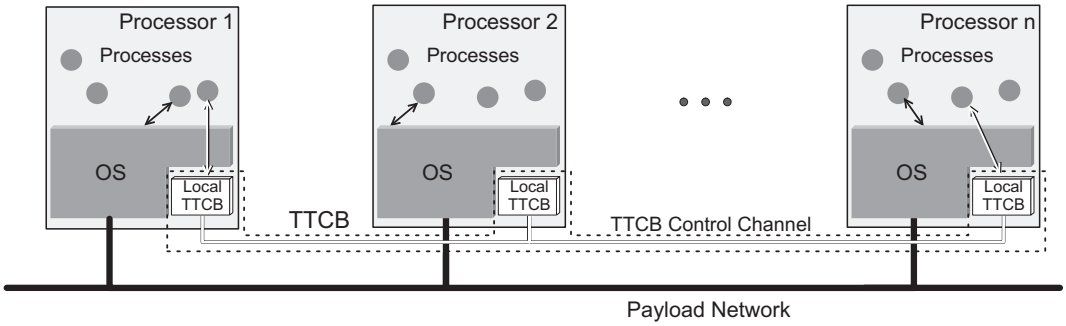

Fig. 1. Architecture of a system with a TTCB.

The architecture of a system with a TTCB is presented in Figure 1. The basic setting is composed of common hosts (e.g., PCs) interconnected by a network (e.g., an Ethernet LAN). This setting is called the *payload system* and is equivalent to what is usually denominated "the system" in the literature.

The TTCB is a distributed component with local parts in hosts – *local TTCBs* – and its own private channel or network – the *control channel*. This component, represented in white in the figure, has a few important characteristics:

- it is secure and can only fail by crashing;
- it is synchronous, capable of timely behavior;
- it provides a small set of services that can be used to implement intrusion-tolerant protocols.

Worm-IT and other applications based on the TTCB can be implemented, for instance, by a set of processes running in the hosts (see figure). These processes communicate mostly through the payload network, but they can use the TTCB as a runtime support to perform some critical steps of their execution. The TTCB is assumed to be secure. This means, that the results returned by any TTCB service are assumed to be correct, i.e., according to the specification of that service.

2.1 TTCB Services

Worm-IT uses only three of the services provided by the TTCB (see (Correia et al., 2002b; Veríssimo and Casimiro, 2002) for a complete list and details). The *Local Authentication Service* establishes a trusted path between a software entity (process, thread, component) and a local TTCB. The service also provides an identification for the entity before the TTCB (*eid*). The *Trusted Timestamping Service* provides globally meaningful timestamps, since the local TTCBs' internal clocks are synchronized.

The *Trusted Block Agreement Service* (TBA) is the main service used by Worm-IT, therefore we describe it in more detail. TBA delivers a value obtained from the agreement of values proposed by a set of software entities running in the hosts. The values are binary blocks with a limited fixed size, and the service is not intended to do all agreement operations in a system, but rather the steps where fast trusted agreement on some control information (e.g., cryptographic hashes) can leverage protocol execution performance.

An entity *proposes a value* for a TBA execution when it calls TTCB_PROPOSE successfully (when no error is returned). An entity tries to *decide a result* by calling TTCB_DECIDE ("tries" because the result may still not be available when the function is called). The *result* is composed not only by a value but also by some additional information described below. Formally, the TBA service is defined by the following properties:

TBA1 Termination Every correct entity eventually decides a result.
TBA2 Integrity Every correct entity decides at most one result.
TBA3 Agreement No two correct entities decide differently.
TBA4 Validity If a correct entity decides *result* then *result* is obtained applying the function *decision* to the values proposed.
TBA5 Timeliness Given an instant *tstart* and a known constant T_{TBA}, the result of the service is available on the TTCB by *tstart*+T_{TBA}.

The interface of the service contains the two functions mentioned:

```
outp ← TTCB_PROPOSE(elist, tstart, decision, value)
```

`outd ← TTCB_DECIDE(tag)`

The parameters of TTCB_PROPOSE have the following meanings. *elist* is a list of the eid's of the entities involved in the TBA. *tstart* is a timestamp that indicates the instant when the TTCB ceases to accept proposals for the TBA, which means, in some sense, the latest instant when the TBA starts to run inside the TTCB. The third parameter, *decision*, is the function used to calculate the result from a list of proposals. The protocols in this paper use two of those functions: TBA_MAJORITY that returns the value proposed by the greatest number of entities; and TBA_RMULTICAST that returns the value proposed by the first entity in *elist*. *value* is the value proposed. An execution of the TBA service is uniquely identified by *(elist, tstart, decision)*.

TTCB_PROPOSE returns a structure *outp* with an error code and a *tag* that is used later to identify the TBA when the entity calls TTCB_DECIDE. This second function returns a structure *outd* with: (1) an error code; (2) the value decided (if the TBA already terminated); (3) a mask *proposed-ok* with one bit set for each entity that proposed the value that was decided; and (4) a mask *proposed-any* with one bit set per entity that proposed any value.

The purpose of the timestamp *tstart* is to prevent malicious entities from postponing TBAs indefinitely by not proposing their values. If all values are available before *tstart*, the TBA is initiated sooner, when the last value arrives to the TTCB. If an entity tries to propose after *tstart*, it gets an error indicating that the value was not accepted, and the tag of that specific TBA. Next, it can call TTCB_DECIDE to collect the decision that was calculated using the values proposed by the other entities. For simplicity, it is assumed throughout the paper that the TTCB can record the output of a TBA for a long time. In practice, the solution is known: the TTCB will have to garbage-collect old results, and a delayed entity that is never able to obtain a decision should be forced to exit the group.

An attacker might attempt a denial-of-service attack against the TBA service simply by making a large number of calls to TTCB_PROPOSE and starting many TBAs. The TTCB uses two mechanisms to prevent this possibility. The first is a resource reservation mechanism that makes the TTCB assign a set of resources to an entity. In the TBA case, this allows an entity to get the guarantee that the TTCB will run on its behalf a number of TBAs per unit of time. The second mechanism is an admission control mechanism that rejects any call to a service that exceeds the resources reserved for the entity.

2.2 TTCB Implementation

The implementation of the TTCB is an issue mostly orthogonal to the present paper. However, we include a short discussion because it helps the reader get a better insight into what is the TTCB, since this type of component is novel. The design

and implementation of a TTCB based on commercial off-the-shelf components was presented in Correia et al. (2002b), and alternatives were discussed.

The TTCB is assumed to be secure, therefore it has to be isolated from the rest of the system, except for a well-defined interface. For the local part, the best way to enforce this isolation is by implementing the local TTCB in a hardware appliance of some kind. A good option is to use a secure processor, like IBM 4758, but hardware for secure applications is currently an important research trend, with several solutions being proposed (Smith, 2004). Another solution would be to use a PC/104 board with its own processor, memory and disk-on-chip. In the currently available COTS-based design, however, a different approach is used. The local TTCB resides inside a real-time kernel, which is hardened in order to be secure. This solution has less coverage of the security assumptions than one based on hardware, but has the advantage of allowing the free distribution of a TTCB by the research community [5] .

Solutions for the implementation of the control channel can range from a dedicated Ethernet LAN (the solution used in the prototype) to some sort of virtual private network (e.g., a set of ISDN, Frame Relay or ATM connections). The LAN can be assumed to be secure if it is a short-range inside-premises closed network, connecting a set of servers from a single institution. For WANs, a combination of cryptographic techniques and the use of parallel channels can be used to prevent most attacks.

The implementation of the local TTCB requires a real-time operating system. The current prototype uses RTAI, a real-time kernel based on Linux and that runs on standard PC hardware. The control-channel has also to be predictable in terms of time behavior. This can be enforced in networks with guaranteed bandwidth by controlling the amount of traffic with an admission control mechanism (Casimiro et al., 2000).

3 System Model

The main idea behind the system model considered in the paper is that it is *hybrid*. Recall the architecture of a system with a TTCB in Figure 1. We make strong assumptions about the TTCB: it is secure and synchronous. However, we do only very weak assumptions about the rest of the system (payload system). This is what we mean by hybrid. This section presents the system model. We skip the assumptions about the TTCB since they were already introduced in the previous section.

The payload system has to most extent unpredictable timeliness, i.e., it is essentially asynchronous. More precisely, we assume no bounds on the communication

[5] Download from: http://www.navigators.di.fc.ul.pt/software/ttcb/

delays, but we make a weak synchrony assumption about delays in hosts: there is an unknown *processors' stabilization time* (PST) such that the processing delays of correct processes are bounded from instant PST onward, and these processing delays bounds are known. This synchrony assumption is weak when compared with the usual partial synchrony assumptions, e.g., in Dwork et al. (1988), because processing delays are much more deterministic than communication delays. Although these time assumptions are weak, consensus in this system model is not bound by the FLP impossibility result (Fischer et al., 1985) since the system is not purely asynchronous, it includes the TTCB that is synchronous.

The payload system is also unpredictable in terms of failure, i.e., it can fail arbitrarily: processors may be intruded, processes and operating systems can be corrupted, and the communication may be attacked.

3.1 Communication Model

This paper presents a group communication system for *groups of processors*, or hosts. Processors rely on channels to hide some of the communication complexity in the payload network. Each pair of processors (p, q) is assumed to be interconnected by a *secure channel* over the payload network. Channels are defined in terms of the following properties:

SC1 Eventual reliability If p and q are correct and p sends a message M to q, then q eventually receives M.

SC2 Integrity If p and q are correct and q receives a message M with $sender(M) = p$, then M was really sent by p and M was not modified in the channel [6].

It is assumed that each pair of correct processors shares a secret key known only by them. The two properties above are simple to implement with these keys. Eventual reliability is achieved by retransmitting the message periodically until an acknowledgment arrives. Message integrity is attained by detecting the forgery and modification of messages using Message Authentication Codes (MACs) (Menezes et al., 1997). A MAC is a cryptographic checksum, obtained with a hash function and a secret key. When p sends a message M to q it concatenates a MAC obtained with their shared secret key to M. When q receives the message it calculates the MAC in the same way and compares it with the incoming MAC. If they are different, the message is either fake or was modified, therefore it is discarded. Reliable channels might also be implemented using the Secure Sockets Layer protocol (Frier et al., 1996).

[6] The predicate *sender(M)* returns the sender field of the message header.

3.2 Processor Failure Modes

A processor is *correct* if it follows the protocol that it is specified to execute. There are several circumstances, however, that may lead to the processor failure. When this happens, a processor can stop working, can delay the transmission of messages, can send messages disregarding the protocol, or can even collude with other malicious processors with the purpose of breaking the protocol. Consider, for now, that V^n is a set with the membership of the group at a certain instant. Worm-IT assumes that at most $f = \lfloor \frac{|V^n|-1}{3} \rfloor$ processors in V^n can fail, i.e., less than one third of the processors. Notice that f is not a constant but is defined for every membership V^n in terms of the number of processors $|V^n|$.

This limit of less than one third of the processors being allowed to fail is common for intrusion-tolerant protocols, since it is the best attainable for consensus to be solvable (Correia et al., 2006). This assumption requires processors to fail independently, something that is not obvious when the source of faults is potentially intelligent (a human attacker) and processors may have all the same vulnerabilities. It is not possible to act on the attacker side of the problem so we have to force the vulnerabilities that might lead to failure to be different, since they usually cannot be entirely avoided. This requires the codes of the system running in the processors to be different in all, the operating systems to be distinct, the root and user passwords to be different, etc (Deswarte et al., 1998; Castro et al., 2003). A solution for the implementation of different code is software diversity obtained using N-version programming (Avizienis, 1985). The approach consists in making independent implementations, possibly in different languages, of software with the same functionality.

3.3 Group Membership Model

Wide-area networks are prone to link failures and other communication fluctuations. These effects can lead to network partitions, i.e., to the virtual separation of the network in several subnetworks that are temporarily unable to communicate. This may cause the temporary division of a group in two or more subgroups. To handle this type of failures, Worm-IT uses a *primary partition* model (Birman, 1997), in which at most one of the subgroups is allowed to make progress. Processors belonging to the rest of the subgroups are eventually removed from the primary partition subgroup. We have to use the primary partition model because the system needs the contribution from at least $2\lfloor \frac{|V^n|-1}{3} \rfloor + 1$ members to decide on a new group membership. Therefore, if the group is partitioned in two or more subgroups, at most one of them will have that number of members and will be able to make progress.

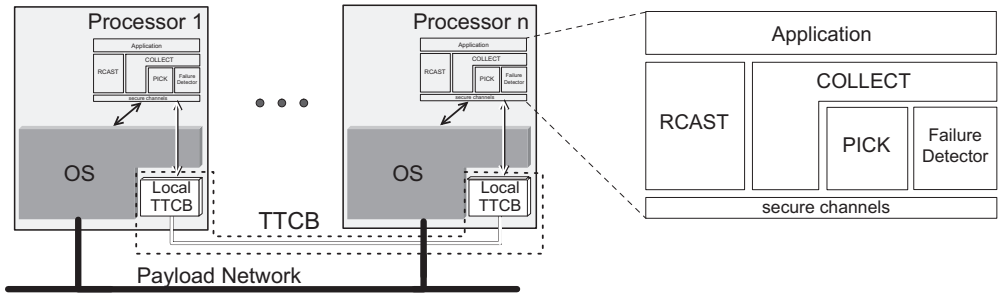

Fig. 2. Architecture of Worm-IT.

There is one more assumption related to the group membership. When a processor wants to join the group, it has to know who are its members, so we assume it can get this information somehow. This issue is further discussed in Section 5.5.

4 The Architecture of Worm-IT

The architecture of Worm-IT is depicted in Figure 2. The membership service is implemented by the COLLECT and PICK protocols. The view-synchronous atomic multicast service is mostly implemented by the RCAST protocol, although it also uses COLLECT and PICK (see Section 6). All protocols use the secure channels on the bottom and there is an Application level on the top, which represents the applications that use Worm-IT to communicate. The Failure Detector module is in charge of detecting the failure of processors (Section 5.6).

COLLECT is a finite state machine that evolves at each processor between two states: NORMAL and AGREEMENT. When a processor joins a group, it enters the NORMAL state. Then, when another processor wants to join or leave, when a processor is suspected to have failed, or when data messages are atomically multicasted, certain *events* are generated and the protocol changes to the AGREEMENT state. In this state, the processors try to agree on membership changes and/or message deliveries by running the PICK protocol. When PICK terminates, the state changes back to NORMAL.

5 Membership Service

A primary partition membership service handles three operations: the addition of members to a group, the removal of failed members, and the removal of members on their own initiative (Chockler et al., 2001). These operations will be called respectively *join*, *remove* and *leave* for short. The failure of a processor is detected in every processor by a *failure detector* module (see Section 5.6).

The Worm-IT membership service generates *views*, i.e., numbered events containing the group membership. A new view is installed whenever the membership is changed due to a member join, leave or removal. A group of processors with a single member is created when the first member joins and installs the first view. A processor P_j sees a view as an array V_j^n containing one entry per each member processor. The index n reflects the n^{th} view of the group. Every processor P_j keeps an array VV_j with the views it has already installed. A view V_j^n is said to be *defined* at processor P_j if $VV_j[n] = V_j^n$, i.e., if that view has been previously installed.

The service guarantees that each correct processor has the same view at every instant of logical time, i.e., after the installation of the same (totally ordered) views in every processor. The membership service executes a protocol defined formally in terms of the following properties (similar to Reiter (1996)):

MS1 Uniqueness If views V_i^n and V_j^n are defined, and processors P_i and P_j are correct, then $V_i^n = V_j^n$.

MS2 Validity If processor P_i is correct and view V_i^n is defined, then $P_i \in V_i^n$ and, for all correct processors $P_j \in V_i^n$, V_j^n is eventually defined.

MS3 Integrity If processor $P_i \in V_i^n$ and V_i^{n+1} is not defined then either at least one correct processor detected that P_i failed or P_i requested to leave. If processor $P_i \in V_i^{n+1}$ and V_i^n was not defined at P_i then at least one correct processor authorized P_i to join.

MS4 Liveness If $\lfloor \frac{|V^n|-1}{3} \rfloor + 1$ correct processors detect that P_i failed or receive a request to join, or one correct processor requests to leave, then eventually V^{n+1} is installed, or the join is rejected.

Uniqueness guarantees that all correct processors in a group see the same membership. Validity ensures that if a view is defined at a processor then the processor is in the view (often called Self-Inclusion property) and that every correct processor in a view eventually installs the view. Integrity prevents malicious processors from removing or adding processors to the group. Liveness ensures that a new view is installed when a number of correct processors detect a failure, or a correct processor wants to join or leave.

The COLLECT protocol handles the three *membership events*, corresponding to the three membership operations mentioned above: join, remove and leave. It also handles one *communication event*, related to the atomic multicast protocol (see Section 6). This section starts by describing the service in terms of generic events $Ev(P_j)$ – event Ev about processor P_j (for instance, Ev can be the event generated by the failure detector in a processor indicating that P_j failed and should be removed). Later, details are given about the correspondence to specific events.

5.1 Example Execution

As a first insight on the execution of the protocol, we will present an example of a processor removal by the membership service (see Figure 3). Initially, the group has four processors, P_1 to P_4. P_4 is malicious and performs malicious actions that are detected by the failure detectors of processors P_1 and P_2. When this happens, P_1 and P_2 multicast a *(INFO, Remv(P_4))* message saying that P_4 should be removed from the group. Even if P_3 does not detect the misbehavior of P_4, when it gets $f + 1 = 2$ messages stating that P_4 should be removed, it knows that at least one correct processor detected the failure, since at most $f = 1$ processors can "lie". Therefore, when P_3 receives the second *(INFO, Remv(P_4))* message, it multicasts the same information.

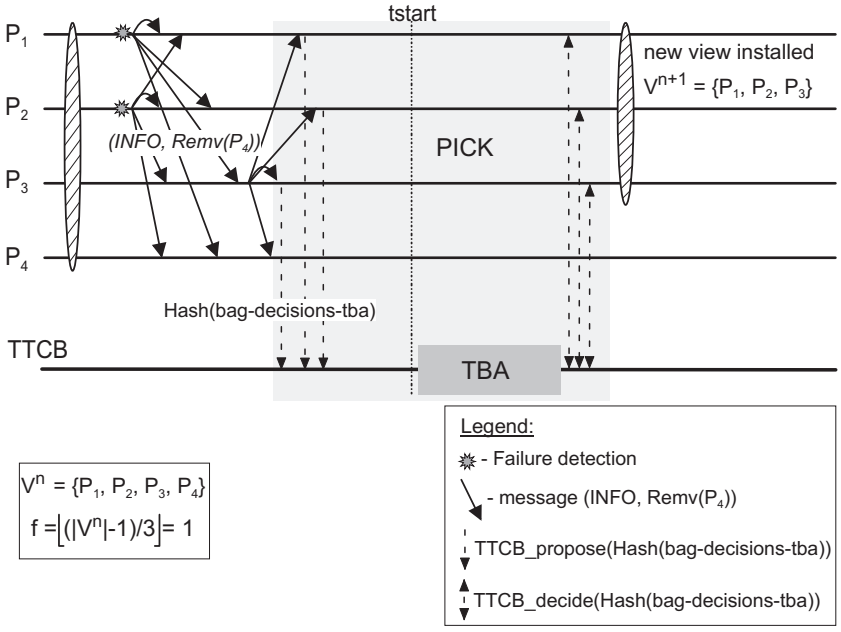

Fig. 3. Membership service example execution.

When a processor receives $2f + 1 = 3$ messages saying P_4 failed it knows that all correct processors will also receive 3 or more messages (justification in the next section). Therefore, it can move to the AGREEMENT state with the confidence that all correct processors will do the same. It can put *Remv(P_4)* in a bag called *bag-decisions*, where it saves all the changes that have to be applied to the current view, also knowing that all correct processors will do the same. The bag is used to store all events that have to be agreed upon by the protocol, but in the case of the membership, this boils down to view changes.

In the AGREEMENT state, the processors execute the PICK protocol. The objective is to make all correct processors decide the same changes to the view. The protocol uses the TBA service of the TTCB to agree on a digest of *bag-decisions-tba*, which is usually identical to *bag-decisions*. In the example, P_1 to P_3 propose identical

digests – they have the same *Remv(P_4)* event in the bag – and TBA returns that digest, since it decides the most proposed value. Next, the new view is installed and P_4 is removed.

5.2 The COLLECT Protocol

The membership service is implemented using two protocols. The basic protocol, COLLECT, is described first (Algorithm 1) and the PICK protocol is presented next (Algorithm 2). Throughout the following discussion, it is assumed that each message carries the current view number. The communication channels only deliver messages that were transmitted in the current view. Messages that were sent in a previous view are discarded and messages for future views are stored for later delivery. The correctness proof of the protocols can be found in Appendix A.

The objective of the first part of the algorithm (lines 6-16) is to guarantee that all (correct) processors get the same events. Whenever a processor finds out that a new event $Ev(P_j)$ has occurred, it sends an INFO message – *(INFO, myid,* $Ev(P_j)$*, valid-tstart-send)* – to all processors in the current view, including itself. A processor can learn about new events in two ways: (1) it "sees" the event by itself, e.g., it detects the failure of P_j (lines 6-10 in the algorithm); or (2) it receives *(INFO, *,* $Ev(P_j)$*, *)* messages from $f + 1$ processors, which mean that at least one correct processor "saw" the event [7] (lines 11-16). In the message, *myid* is the processor identifier and *valid-tstart-send* is a timestamp (discussed later). Processors put the INFO messages that arrive in the bag *bag-info* (lines 11-12). Function *count(*$Ev(P_j)$*, bag-info)* counts the number of INFO messages with $Ev(P_j)$ received from different processors (line 13).

When a processor receives *(INFO, *,* $Ev(P_j)$*, *)* messages from $2f + 1$ different processors (line 17), it knows that all correct processors will receive at least that number of messages because: (1) if the processor received $2f + 1$ messages then every correct processor will eventually receive at least $f + 1$ messages (since at most f processors can fail); (2) when these correct processors receive these $f + 1$ messages they will also multicast (line 16). Therefore, when a processor receives $2f + 1$ INFO messages about $Ev(P_j)$ it can put $Ev(P_j)$ in *bag-decisions*, with the confidence that all correct processors will eventually do the same (lines 17-18). If the processor is still in the NORMAL state it goes to the AGREEMENT state (lines 19-21), since the first condition in line 19 is always true when Ev is a membership event. When a processor executes the PICK protocol, it passes an argument with the smallest *valid-tstart-send* that was received in INFO messages, which is returned by the function *smallest-tstart* (line 21). The meaning of *valid-tstart-send* will be clarified in the next section.

[7] The star '*' is a wildcard that indicates any value.

Algorithm 1 The COLLECT protocol.

```
 1: INITIALIZATION:
 2: bag-info ←∅;                                          {bag with INFO messages}
 3: bag-decisions ←∅;                          {bag with view changes and messages to deliver}
 4: valid-tstart-send ←⊥;                  {valid tstart to send in INFO messages in this view}
 5: state ←NORMAL;                                                  {protocol state}

 6: when Ev(P_j) do                                                  {handle an event}
 7:    if (I did not multicast (INFO, myid, Ev(P_j), *) in this view) then
 8:       if (valid-tstart-send = ⊥) then
 9:          valid-tstart-send ←next-valid-tstart();
10:       multicast (INFO, myid, Ev(P_j), valid-tstart-send);

11: when M = (INFO, sender-id, Ev(P_j), valid-tstart) received do    {handle INFO msg}
12:    bag-info ←bag-info ∪ {M};
13:    if (count(Ev(P_j), bag-info) ≥ f+1) and (I did not multicast (INFO, myid, Ev(P_j),*)
       in this view) then
14:       if (valid-tstart-send = ⊥) then
15:          valid-tstart-send ←next-valid-tstart();
16:       multicast (INFO, myid, Ev(P_j), valid-tstart-send);
17:    if (count(Ev(P_j), bag-info) = 2f+1) then
18:       bag-decisions ←bag-decisions ∪ {Ev(P_j)};
19:       if ((Ev ≠ Datamsgs) or (count(Datamsgs, bag-decisions) = WM)) and (state
          = NORMAL) then
20:          state ←AGREEMENT;
21:          execute pick( smallest-tstart(bag-info) );

22: when bag-decisions-picked = pick(tstart) returned do          {handle end of PICK}
23:    deliver messages corresponding to Datamsg events in bag-decisions-picked or-
       dered by mid.tstart; remove them from bag-data-msgs and the events from bag-
       decisions-picked;
24:    if (there are view change events in bag-decisions-picked) then   {install new view}
25:       add/remove processors in view change events in bag-decisions-picked from view;
26:       view-number ←view-number + 1;
27:       bag-info ←∅; bag-decisions ←∅; valid-tstart-send ←⊥; bag-data-msgs ←∅;
28:       send system state to new members;
29:    state ←NORMAL;
```

When the PICK protocol decides a value, i.e., a set of view changes, some housekeeping is performed and the state goes back to NORMAL (lines 22-29). If new members join the group, they may have to be informed about the system state, including the current membership (line 28).

An event is only considered for agreement in the PICK protocol when $2f + 1$ or more processors have shown that they know about it. Until this quorum is reached, the event is simply stored for later processing. Consequently, there might be some events that still may need to be dealt with when a new view is installed. The solution that was chosen for this problem requires that these events be re-issued in the next

view(s), until they are eventually processed. This solution is relatively simple to implement because it only requires that processors re-send their requests (in case of joins or leaves), or that the failure detector re-indicates the failure of a processor.

5.3 The PICK Protocol

The previous section explains how processors decide to engage in the PICK protocol (Algorithm 2). PICK runs as a series of executions of the TTCB TBA service, each one trying to agree in which way the current view needs to be updated. In the best case, which corresponds to the most common scenario, only one TBA is executed, as illustrated in the example of Figure 3. Later, we will discuss why several calls to the TBA might be needed.

The core of the protocol is presented in lines 8-19. Each processor goes on proposing to successive TBAs the changes it thinks have to be applied to the current view, i.e., the processors to add/remove. These updates are put in *bag-decisions* by the COLLECT protocol (see previous section). Then, PICK copies *bag-decisions* to *bag-decisions-tba* (line 9), since *bag-decisions* can be modified by COLLECT during the execution of the TBA service. PICK gives TBA a hash of *bag-decisions-tba* rather than the actual bag (function *Hash* in line 12) because the TTCB has a limit for the size of the values that it accepts. A *collision-resistant hash function* produces a fixed size digest of its input with the guarantee that it is computationally infeasible to discover another input that gives the same output (Menezes et al., 1997). The current implementation of the TTCB bounds the values to 160 bits, which is enough for standard hash functions like SHA-1. Notice that two processors obtain the same hash of *bag-decisions-tba* only if their two bags have the same bit-by-bit content. This goal is achieved by representing bag data in some canonical form, so that bags with the same content are bitwise identical.

TBA gets the values given by the processors (line 12), then chooses and returns the most frequently proposed value (TBA_MAJORITY decision function). It also returns a mask *proposed-ok* indicating which processors gave the value that was decided (line 14). Processors go on engaging in TBAs until a set with at least $2f+1$ elements proposed the same (line 19). This loop is assured to terminate because all correct processors (at least $2f + 1$) will eventually get the same values in *bag-decisions-tba* (see previous section); therefore, they eventually propose identical hashes and this is the decision value (since they are the majority). TBA is executed inside the TTCB so its results are reliable and all correct processors receive the same output.

If a processor has the *bag-decisions-tba* corresponding to the chosen hash (line 21), it sends that bag to the processors that did not propose the correct hash, to ensure that all correct processors get the changes to the view (line 22). These processors, for instance, might have more events in their *bag-decisions-tba* than the others.

Algorithm 2 The PICK protocol.

```
 1: FUNCTION pick(tstart)
 2: hash-v ←⊥;                                          {hash of the value decided}
 3: bag-msgs-picked ←∅;                                 {bag for PICKED messages received}
 4: elist ←list with eid's of processors in V^{view−number} in ascending order;
 5: if (tstart < last-tstart) then
 6:    tstart ←last-tstart;
 7: data-msgs-deadline ←⊥;

 8: repeat
 9:    bag-decisions-tba ←bag-decisions;        {bag-decisions is shared with COLLECT}
10:    if (data-msgs-deadline ≠ ⊥) then
11:       remove from bag-decisions-tba all events Datamsg with mid.tstart > data-msgs-
          deadline;
12:    outp ←TTCB_PROPOSE(elist, tstart, TBA_MAJORITY, Hash(bag-decisions-tba));
13:    repeat
14:       outd ←TTCB_DECIDE(outp.tag);
15:    until (outd.error ≠ TBA_RUNNING);
16:    if (data-msgs-deadline = ⊥) and (2f + 1 processors proposed any value) then
17:       data-msgs-deadline ←tstart;
18:    tstart ←tstart + T_{retry};
19: until (at least 2f + 1 processors proposed the value that was decided);
20: last-tstart ←tstart;
21: if (outd.value = Hash(bag-decisions-tba)) then
22:    multicast (PICKED, myid, bag-decisions-tba) to processors not in outd.proposed-ok
       and not being removed;
23:    return bag-decisions-tba;                       {terminates the algorithm}
24: else
25:    hash-v ←outd.value;

26: when M=(PICKED, *, *) received do                  {message with decisions picked}
27:    bag-msgs-picked ←bag-msgs-picked ∪ {M};

28: when (hash-v ≠ ⊥) and (∃_{M∈bag−msgs−picked} : Hash(M.bag-decisions) = hash-v) do
29:    return M.bag-decisions;                         {terminates the algorithm}
```

Therefore, unless they are informed about the necessary updates, they do not know how to move to the next view. The processing of these messages is done in lines 24-29.

The protocol uses a timestamp *data-msgs-deadline* to exclude some events from *bag-decisions-tba* (lines 7, 10-11, 16-17). This is used only by VSAM so we leave this discussion for Section 6.

The parameter tstart

Now, we delve into the details of the parameter *tstart*, which is passed to PICK (line 1) and used for the successive TBAs (line 12). We start by introducing two notions:

Participation in a TBA A processor *participates* in an execution of the TTCB TBA service (or *in a TBA*) defined by *(elist*$_1$*, tstart*$_1$*, decision*$_1$*)* iff it calls TTCB_PROPOSE*(elist*$_1$*, tstart*$_1$*, decision*$_1$*, *)*.

Active participation in a TBA A processor *participates actively* in a TBA defined by *(elist*$_1$*, tstart*$_1$*, decision*$_1$*)* iff it calls TTCB_PROPOSE*(elist*$_1$*, tstart*$_1$*, decision*$_1$*, *)* before *tstart*$_1$ [8] .

For a correct processor to know which set of view changes (*bag-decisions-tba*) should be applied to the current view it is not required to participate in all TBAs that are executed: it only needs to participate in the TBA that satisfies the condition in line 19. Let us call this TBA the *last TBA*. This TBA provides the processor a hash of the *bag-decisions-tba* that has to be applied. Therefore, it becomes capable of selecting the correct *bag-decisions-tba*: either its own (lines 21 and 23) or one of the various that might arrive (lines 26-27)

The participation of a processor in a last TBA does not have to be *active*. If it is not active, the proposal from this processor is not accepted by the TTCB, but the processor obtains the *tag* of that TBA execution, and eventually gets the decision (lines 13-15).

How do we guarantee that all correct processors *participate* in the *last TBA*? First, we define a discrete set of values that can be used for *tstart*:

Valid tstart A timestamp is called a *valid tstart* if it is in the set $\{\forall_{k \in N}, k * T_{retry}\}$, where T_{retry} is the interval between valid *tstarts*.

The value T_{retry} must be higher than the maximum delay for one execution of lines 8-19 after PST. However, the value also involves a tradeoff: if T_{retry} is too low, on average more TBAs will be used to reach agreement but PICK will usually terminate faster; if T_{retry} is too high, on average the contrary will happen: less TBAs but PICK might take longer to terminate. The function *next-valid-tstart()* returns the next *valid tstart* timestamp after the present instant.

Second, we have to guarantee that when a processor enters the loop in line 8, the value of *tstart* is less than or equal to the *tstart* of the last TBA (since the loop increases *tstart*). This condition is ensured if processors initiate the PICK protocol with the smallest *tstart* of the INFO messages (lines 8-9, 14-15, and 21 of Algorithm 1). The following reasoning can be used to understand why this is true. In a

[8] The admission control mechanism has also to accept the call to TTCB_PROPOSE for the participation to be active.

PICK execution, the first TBA in which a processor participates has a *tstart* greater than or equal to the *tstart* that was passed as argument in line 1, due to lines 5-6. The last TBA has the participation of $2f + 1$ or more processors, so the *tstart* of the last TBA is greater than or equal to the initial *tstarts* of all the processors that actively participated in that TBA. On the other hand, a processor passes to PICK the smallest *tstart* from the $2f+1$ INFO messages that were received (Algorithm 1, line 21). Consequently, since the intersection of the set of processors that actively participated in the last TBA and the set of processors that sent the INFO messages has at least one element (f processors can be malicious and "lie"), the *tstart* used for the first TBA in each correct processor is smaller than or equal to the *tstart* of the last TBA. This is precisely what we wanted to show.

A malicious processor could attempt to delay the protocol by providing an INFO message with a "very small" valid *tstart*, i.e., a timestamp that had passed a long time ago. To prevent this type of attack, the repeat loop is always initiated with a *tstart* larger than the *tstart* of the previous PICK execution (lines 5-6). The reader should notice that this attack would not cause any incorrect behavior of the protocol – it would simply delay the execution.

A final discussion is due on the possibility of having to run several agreements (TBAs) inside the TTCB in order to make a single agreement outside it. The issue has to do with the intrinsic real-time nature of the TTCB and the TBA service, and the asynchrony of the rest of the system. When a processor calls TTCB_PROPOSE it provides a *tstart*, i.e., a timestamp that indicates to the TTCB the instant when no more proposals are accepted for the TBA identified by the arguments *(elist, tstart, decision)*. The processor that calls TTCB_PROPOSE is in the asynchronous part of the system, so *tstart* is for it a mere integer number, and therefore we can never assume that the processor will call TTCB_PROPOSE before instant *tstart*, regardless of the value of this parameter. The consequence to the PICK protocol is that in each round any number of processors may not be able to propose before *tstart*. This is the reason why the protocol may have to run several rounds and call successive TBAs, until enough processors manage to propose before *tstart*, i.e., until the condition in line 19 is satisfied. The safety of the protocols asynchronous operation is thus ensured, despite its use of a synchronous function (TBA). The liveness is ensured by the assumption on the processors' stabilization time (Section 3), which guarantees that eventually enough hosts will be able to call TTCB_PROPOSE before *tstart*, and the PICK protocol will terminate (lines 12 and 19). The increment to *tstart* in line 18 must be higher than the eventual maximum delay for one execution of lines 8-19, but we assumed there is a known bound for these delays from PST onward (Section 3). Notice that the safety of the system does not depend of existing a PST, only its liveness.

5.4 Processor Leave

The COLLECT protocol was described in terms of generic events $Ev(P_j)$. Now let us see the corresponding membership events, starting with the event related to the removal of a member on its own initiative.

A processor P_j can decide to leave a group for several reasons, for example, because the user wants to shutdown its machine. This decision is usually taken by a higher-level software module (*Application* in Figure 2). When that happens the processor multicasts a message *(LEAVE, myid)* to all processors in the group (including itself). The reception of this message is the leave event *Leave(*P_j*)*. This event is then handled by the COLLECT protocol as described in the previous sections. Notice that a malicious processor cannot remove a correct processor P_j by sending a *(LEAVE,* P_j*)* message due to Integrity property of the communication channels (see Section 3.1).

5.5 Processor Join

In the crash fault model, a processor that wants to join a group has simply to find a *contact* with the information about the group membership. The contact can be any member of the group or some kind of third party. In the Byzantine fault model, the problem is more complex since individual processors or other entities may provide erroneous information. For instance, if a processor that wants to join asks the current view from a malicious processor P_i, then P_i could return a group composed exclusively of malicious members. Therefore, the implementation of the join operation in an arbitrary failure environment requires the solution of two sub-problems: first, it is necessary to determine who should be contacted; second, if several answers are received with the information about the group, it is essential that the correct one be selected.

There are two generic solutions for both problems. Either one considers the existence of a reliable (i.e., trusted) well-known source, or one has to contact a set of processors and assume that at least a majority of $2f + 1$ of them are correct for f faulty. Specific examples of these methods are:

- The system administrator manually provides a list of the current group members.
- There is a trusted third party server that always returns correct results.
- There is a set of n' potential member processors from which no more than f' can fail ($n' \geq 3f' + 1$). Each processor in the set, even if not a current member of the group, keeps membership information and provides it when requested.

Independently of the selected approach, it is assumed that a joining processor P_j manages to obtain the current view of the group, V^n. Then, P_j multicasts a message

(REQ_JOIN, myid, auth-data) to all processors in V^n. *auth-data* is application dependent authorization information that is independent of the membership protocol. Therefore, when the REQ_JOIN message arrives, the protocol upcalls the application asking for the approval of the new processor (passing *auth-data* as parameter). If P_j is accepted, a join event *Join(P_j)* is generated for further processing. Later, when the new view is installed, P_j gets the group state (Algorithm 1, line 28). The processor has to wait for $f+1$ identical copies to know it received the correct state.

5.6 Processor Removal and Failure Detection

The *failure detector* module in a processor determines if other processors have failed, and produces events *Remv(P_j)* which are then handled by the membership service (Algorithms 1 and 2). Although the design of a Byzantine failure detector is not the subject of this paper, we provide some insights about its implementation through the rest of this section.

Byzantine failure detectors have to detect different faulty behaviors in the system, ranging from accidental crash to malicious actions. Detectors for malicious faults are hard to develop because they have to be designed, at least in part, in a way that depends on the protocols being used by the processors monitored (Doudou et al., 2002; Baldoni et al., 2003). They have to know and understand the expected behaviors of these protocols, otherwise, some types of attacks cannot be detected. For this reason, they should look for the following activities during the execution of the protocols (some of these ideas are borrowed from Malkhi and Reiter (1997); Doudou et al. (2002); Kihlstrom et al. (2003)):

- Determine if a processor completely stops interacting, either because it crashed or because it is malicious.
- Find out if a non-crashed processor is silent for some part of a protocol or application execution, i.e., if it does not send some expected messages but it continues to send others. For example, in our particular case, a processor that does not send a message *(INFO, myid, Ev(P_j), *)* after receiving $f+1$ INFO messages with the same event.
- Determine if a processor sends incorrectly formed or out-of-order messages. For example, a processor replays some previously sent message.
- Establish if a processor sends unexpected messages or messages with incorrect content. For instance, a processor that sends a *(INFO, id, Ev(P_j), *)* with an *id* different from its own.
- Find out if a processor is being externally attacked and intruded. The output of an Intrusion Detection System could be used as an indication of the intrusion.

Although detecting these problems would be desirable, we discussed in the introduction that theoretically it is impossible to detect crashes in asynchronous systems

since it is not possible to differentiate when a processor is very slow or crashed. Moreover, we also discussed that doing so in practice introduces vulnerabilities in the system since it allows malicious attackers to remove a correct process simply by forcing it to slow down and causing an erroneous detection of a crash. This reasoning might be expanded for other kinds of malicious faults that might be wrongly attributed to a correct process. Therefore, we have to be conservative and design a failure detector that satisfies the following property of the taxonomy proposed by Chandra and Toueg (1996):

Strong Accuracy: No process is suspected before it fails.

This involves giving up to detect failures that involve time, due to the asynchrony of the payload system. This is a limitation of the asynchronous model that cannot be circumvented. These kinds of faults that involve time have to be detected administratively, possibly with the assistance of a failure detector that provides only failure suspicions.

The mentioned taxonomy classifies failure detectors also in terms of *completeness*. However it is clear that some malicious faults cannot be detected and that no completeness can be guaranteed.

Notice that the membership and the atomic multicast protocols do not depend on the failure detector to make progress (on the contrary of Rampart (Reiter, 1996) and SecureRing (Kihlstrom et al., 2001)). If the failure detector does not detect a failure, for instance because it is not able to detect a given class of failures, the membership will not remove the corresponding processor from the view. However, it will still be able to behave correctly, i.e., to go on installing new views and delivering messages. Therefore, Worm-IT does not suffer from the same problem as those other works.

6 View-Synchronous Atomic Multicast

This section presents the *View-Synchronous Atomic Multicast protocol (VSAM)*. This protocol provides a view-synchronous semantics, i.e., it guarantees that all correct group members deliver the same messages in the same view (Birman and Joseph, 1987b,a) [9] . Group communication usually involves a set of reliable multicast primitives with different order properties. VSAM orders messages in total order, i.e., all correct processors deliver the messages in the same order. The protocol is defined in terms of the following properties:

[9] View synchrony was originally designated *virtual synchrony* since the objective was to give the idea that events occurred synchronously, in the same order, in all processes/processors (Birman and Joseph, 1987a). This intuition did not fit well with a second generation of group communication systems that supported partitionable groups, therefore the more general definition and designation of *view synchrony*.

VSAM1 Validity If a correct processor multicasts a message M, then some correct processor in *group(M)* eventually delivers M.

VSAM2 Agreement If a correct processor delivers a message M, then all correct processors in *group(M)* eventually deliver M.

VSAM3 Integrity For any message M, every correct processor 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)*.

VSAM4 Total order If two correct processors in *group(M)* deliver two messages M_1 and M_2 then both processors deliver the two messages in the same order.

VSAM5 View synchrony If two correct processors in *group(M)* install views V^n and V^{n+1} then both processors deliver the same messages in view V^n.

There are several similar definitions of view synchrony in the literature. The definition used in here is inspired in Chockler et al. (2001). The predicate *group(M)* indicates the members of the group in the view in which the message is eventually delivered, since the message does not have to be delivered in the view in which it was initially multicast. *sender(M)* is the sender of the message.

6.1 The VSAM Protocol

This section describes the VSAM protocol. The protocol assumes that at most $f = \lfloor \frac{|V^n|-1}{3} \rfloor$ processors can fail in a given view V^n. VSAM uses the protocol RCAST (Algorithm 3) to guarantee essentially that all processors deliver the same messages (properties VSAM1-3) and the protocols COLLECT and PICK to guarantee total order and view synchrony (properties VSAM4-5).

RCAST uses the payload network to multicast messages and the TBA service to distribute unique, reliable, hashes of these messages. When a processor wants to atomically multicast a message with some *data*, RCAST builds a message *(DATA, elist, tstart, data)* and gives its hash to the TBA service (lines 7-8). A message of the VSAM protocol is uniquely identified by *mid = (sender-eid,tstart)* [10] . The decision function is TBA_RMULTICAST so the value returned by the TBA is the value proposed by the first entity in *elist* (Section 2.1), i.e., the hash of the message provided by its sender. When the sender defines a *tstart* for the TBA as the current instant plus a delay T_{sched} (line 7) it has no guarantee that it will manage to propose for the TBA before that *tstart* (line 8). However, the assumption in Section 3 guar-

[10] This uniqueness requires the lists with processor identifiers (*elist*) to be in a canonical form: the first *eid* is the sender's, and the others are in ascending order. A brief justification for the uniqueness of *mid*: the sender uses the TTCB TBA service to give all processors a hash of the message; an execution of the TBA is uniquely identified by *(elist, tstart, decision)* (Section 2.1) and *decision* is hard-coded in the protocol *(*TBA_RMULTICAST*)*; the *sender-eid* identifies a single *elist* in canonical form in a view; therefore it is not possible to send another message in the same view with the same id.

antees that this will eventually happen if T_{sched} is higher than the maximum delay for one execution of lines 6-9 after PST. After successfully proposing, the sender multicasts the DATA message (line 10).

Algorithm 3 RCAST protocol.

```
 1: INITIALIZATION:
 2: for all mid do hash-msg[mid] ←⊥; {table with info about DATA messages received}
 3: bag-data-msgs ←∅;                          {bag with the messages to be delivered}

 4: when VSAM-multicast(data) is called do                                  {SENDER}
 5:    elist ←all processors in current view in canonical form;
 6:    repeat
 7:       M ←(DATA, elist, TTCB_getTimestamp() + T_sched, data);
 8:       outp ←TTCB_PROPOSE(M.elist, M.tstart, TBA_RMULTICAST, Hash(M));
 9:    until (outp.error ≠ TSTART_EXPIRED);
10:    multicast M to all processors in current view;
11:    bag-data-msgs ←bag-data-msgs ∪ {M};
12:    generate event Datamsg(mid); terminate;  {terminates protocol for message [mid]}

13: when DATA message M with identifier mid is received do                 {RECIPIENTS}
14:    if (M.elist[0] ≠ my-eid) then          {handle message only if I'm not the sender}
15:       if (hash-msg[mid] = ⊥) then
16:          outp ←TTCB_PROPOSE(M.elist, M.tstart, TBA_RMULTICAST, Hash(M));
17:          repeat
18:             outd ←TTCB_DECIDE(outp.tag);
19:          until (outd.error ≠ TBA_RUNNING);
20:          if (outd.error ≠ OK) then
21:             return;
22:          hash-msg[mid] ←outd.value;
23:       if (Hash(M) = hash-msg[mid]) then
24:          multicast M to processors in the view except those in mask outd.proposed-ok;
25:          bag-data-msgs ←bag-data-msgs ∪ {M};
26:          generate event Datamsg(mid); terminate;
```

When a recipient receives a DATA message *M* for the first time, it participates in the corresponding TBA to get the hash of the message (lines 13-19). If *M* corresponds to the hash given by the TBA (line 23), the recipient becomes aware that the message is indeed the message sent, so it resends it to all processors which did not provide the correct hash to the TBA (line 24). This guarantees that all correct processors eventually receive the message.

The RCAST protocol does not enforce the properties of Total Order and View Synchrony, therefore it does not deliver the messages to the *Application* module, but saves them in *bag-data-msgs* instead (lines 11 and 25).

Properties VSAM4-5 are enforced by the COLLECT and PICK protocols. The membership service handles three types of events: *Join, Remv* and *Leave*. RCAST generates a fourth type of event when it is ready to deliver a message identified by

mid: *Datamsg(mid)* (Algorithm 3, lines 12 and 26). This event is handled by Algorithm 1 mostly in the same way as the view change events: an INFO message is sent (lines 6-10) and when there are $2f + 1$ INFO messages it is inserted in a bag and PICK can be started (lines 11-19). A difference in relation to the membership events is that a single *Datamsg* event does not start PICK, to avoid running one or more TBAs for each DATA message. PICK is started when a certain condition is satisfied. The code shows one possibility: PICK starts when a certain watermark number of DATA messages WM are ready to be delivered (line 17). Another possibility would be to start PICK when a certain time passed from the last delivery. The code can be easily modified to test this condition or a combination of both.

At this stage, it should be clear that PICK is used with two different but related purposes. The first is to make all processors agree on the view changes to do to the current view (see Sections 5.2 and 5.3). The view synchrony property states that all correct processors deliver the same messages in a view. Therefore, when PICK decides on view changes it also decides on the non-delivered messages still to be delivered in the view (if any).

The second purpose of PICK is to agree on messages to be delivered *when there are no view changes*. The objective is to satisfy the total order property. In each execution of PICK all processors agree to deliver the same set of messages. Then, these messages are ordered according to their *tstart* and delivered in order (Algorithm 1, lines 22-23).

The deadline *data-msgs-deadline* in Algorithm 2 requires some discussion. In Section 5.3 we mentioned that this deadline was used only for VSAM. The issue is that the constant reception of new *Datamsg* events might prevent PICK from terminating since *bag-decisions* would go on changing indefinitely. The same cannot happen with view change events because their number is limited [11] . The solution for this problem is the definition of a deadline *data-msgs-deadline* for each PICK execution (Algorithm 2, lines 7, 11, 16-17). Any message with *tstart* greater than *data-msgs-deadline* is not considered for that execution of PICK (line 11). *data-msgs-deadline* is the first *tstart* in which at least $2f + 1$ processors propose any value for a TBA (lines 16-17), and it was proved in Section 5.3 that all correct processors *participate* in that TBA.

The correctness proof of the protocol is sketched in Appendix A.

[11] The processors that can be allowed to join have to be known in order to be authenticated, therefore their number is limited. The number of processors that can leave or be removed is at most the number of members.

7 Performance Evaluation

7.1 Membership Service Performance

The performance of the membership service was evaluated using a COTS-based TTCB (Correia et al., 2002b). The experiments were performed on a system with six 450 Mhz Pentium III PCs with 192 Mbytes of RAM. The payload and control networks were two 100 Mbps switched Fast-Ethernet LANs. The code was implemented in C and compiled with *gcc*. The versions of software used were TTCB 1.11, RTAI 24.1.10 and gcc 2.96. The MD5 hash function was used both by the PICK protocol (Section 5.3) and the MACs used to protect the communication (Section 3.1). The messages were multicast using IP multicast. Since the maximum number of PCs was limited to six, it was necessary to set $f = 1$. The value used for T_{retry} was 14 milliseconds, a value slightly above the maximum time TBA takes to run in the current implementation, $T_{TBA} = 13$ milliseconds. Each measurement was repeated at least 1000 times, and the figures present average values.

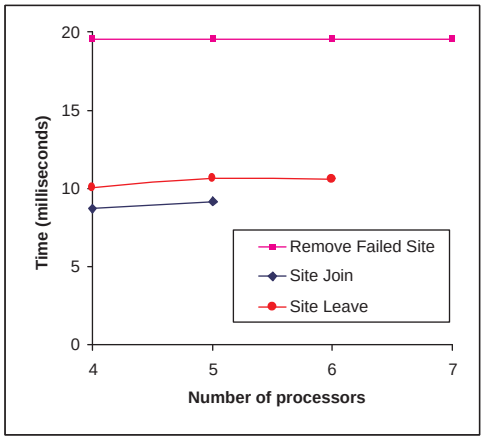

Fig. 4. Average times to install a new view with the operations remove, join and leave.

The results of the experiments are presented in Figure 4. There were *three experiments*. The first quantified the time to *remove a failed processor* that stopped interacting, either because it crashed or was corrupted. The failure detection was simulated multicasting a short message. The second experiment assessed the time for a processor to *join*. The processor multicast a REQ_JOIN message to all others, waited for them to install a new view and to get the state transfer from $f + 1 = 2$ processors. No authorization scheme was used. The third experiment evaluated the time for a processor to *leave* the group. The processor multicast a LEAVE message to the group and measured the time until the new view was installed without itself. For the first experiment, the values presented in the figure are the average of the times measured by all correct processors. For the second and third experiments, the times were assessed respectively by the processors that joined and left.

The protocol spends most of the time in the following operations: calculating the MACs for the INFO messages; exchanging these messages; and executing TBAs. The time taken executing TBAs is the most important since no public-key cryptography is utilized. A consequence is that the performance does not change much with the number of processors (see figure). Most experiments required a single execution of the TBA service. In 3.4% of the experiments some processors tried to propose a value to the TBA after *tstart*, and in a few of these cases two TBAs had to be executed. The join and leave experiments have similar execution times. The time to remove a processor from the group is almost the double of the other two. The reason for this behavior is a subtlety about the TBA service. The stopped processor did not engage in the PICK protocol, therefore not all the processors participated actively in the TBAs executed. When this happens, the TBA starts only by *tstart*, which is usually later than when all processors participate actively (Correia et al., 2002b). In the join and leave experiments all processors participated in the TBAs, usually actively, so the TBAs started and terminated earlier.

Currently, we are aware of only three other implementations of membership services for systems that might experience Byzantine faults: Rampart (Reiter, 1996), ITUA (Ramasamy et al., 2002) and SecureRing (Kihlstrom et al., 2001). The system model and experimental settings used in the evaluations of these services were different from ours, consequently, it is quite difficult to make a comparison among the various performance results. Nevertheless the performance obtained with WormIT is at least of the same order of magnitude.

7.2 VSAM Performance

The performance of VSAM was evaluated using the same setting as the membership service. The evaluation consisted in three sets of experiments, each one using at least 1000 messages. The experiments measured the average delivery time (latency) and the sustainable throughput of the protocol. The experiments were performed in rounds. Each round started with the processors multicasting and receiving a set of messages using RCAST. Then, the processors executed the PICK protocol, delivered the messages and started another round. The prototype did not send messages during the execution of PICK.

The first set of experiments evaluated the performance of VSAM with 4 processors, a single sender, no failed processors and watermarks (WM, the number of DATA messages that causes PICK to start) ranging from 1 to 25 messages (Figure 5). The messages payload was 100 bytes long (headers not included). The figure shows a strong increase in the throughput that slows down around $WM = 5$ messages. When the watermark is low ($WM < 5$), the time taken by the PICK protocol to agree on the messages to deliver is comparable to the time taken to send the messages, i.e., the time spent in the first part of the round, so each additional value of

WM causes a considerable improvement of the throughput. When the watermark is higher (e.g., $WM > 10$), the time used by the PICK protocol is negligible in face of the time taken to send messages, so the throughput improves slowly with WM. The average latency, on the contrary, increased steadily with the watermark, since the system delivers the messages when PICK terminates. When the watermark becomes higher, the time the first messages sent wait for PICK to terminate also becomes higher.

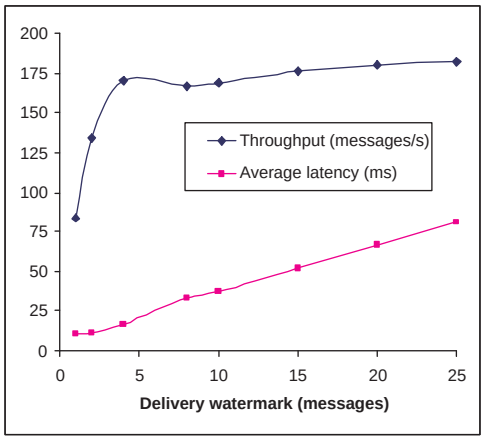

Fig. 5. VSAM performance variation with WM (4 processors, one sender, 100 bytes messages).

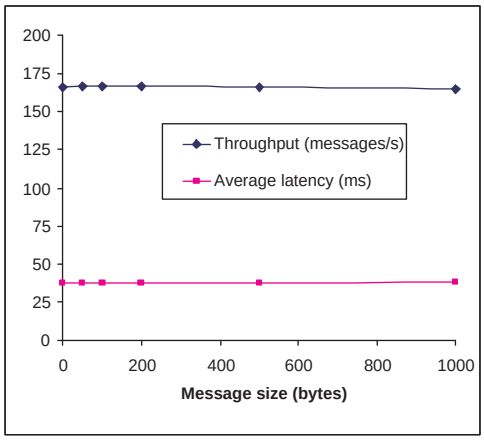

Fig. 6. VSAM performance with different message sizes (4 processors, one sender, $WM = 10$).

The second set of experiments measured the performance of the protocol with different message sizes (see Figure 6). The value selected for the watermark was $WM = 10$, which the first set of experiments indicated to be a good tradeoff between throughput and latency. The number of processors was 4, there was a single sender and no failed processors. The figure allows us to conclude that the size of the message does not affect considerably either the throughput or the latency. Since the weight of the time to transmit the messages is considerably inferior to the time to run the dissemination of the messages done by the COLLECT protocol (lines 10

and 16) and the execution of the PICK protocol.

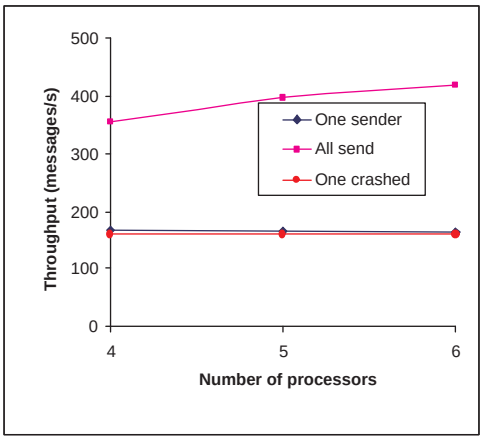

Fig. 7. VSAM throughput with one sender, all processors sending, and one sender with one silent processor ($WM = 10$, 100 bytes).

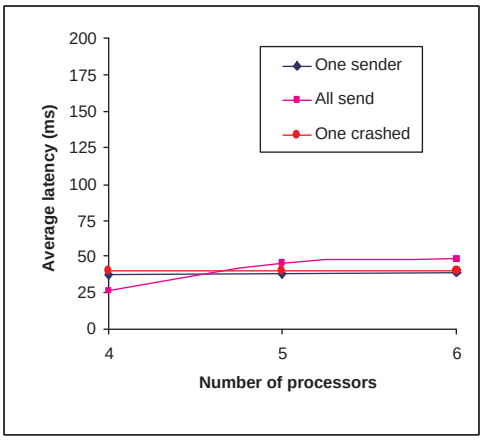

Fig. 8. VSAM average latency with one sender, all processors sending and one silent processor ($WM = 10$, 100 bytes messages).

The third set of experiments compared the performance of the protocol with 4 to 6 processors in three different situations. The results are shown in Figures 7 and 8. The first case was the same as in the first experiment, i.e., no failed processors, a single sender and $WM = 10$ messages. In the second situation *all* processors sent messages, instead of only one. The watermark was twice the number of processors, i.e., respectively 8, 10 and 12. In the third case there was a single sender but one processor was silent, so it did not participate in the protocol. The idea was to understand what happens when a processor crashes or there is an intrusion and it does not run the system code any longer.

The first conclusion from the experiments is that the protocol has a higher throughput when all processors send, something that usually happens (see, e.g., (Reiter, 1994)). The second conclusion is that a crashed processor does not affect the performance of the protocol due to the nature of the COLLECT and PICK protocols

that make decisions in a distributed way (instead of in a centralized way like Rampart). The throughput is in fact slightly better since the silent processor does not multicast either DATA or INFO messages. The latency is 2 to 3 ms worse because the TBA takes longer to run when at least one processor does not propose (Correia et al., 2002b). The same happened for the membership service.

There are some numbers available for the performance of intrusion-tolerant view-synchronous atomic multicast protocols in the literature: Rampart (Reiter, 1994) and its more recent implementation by project ITUA (Ramasamy et al., 2002). Once more the performance obtained for Worm-IT was comparable to those systems.

7.3 Performance Comparison

Comparing the performance of different systems using experimental results is difficult since the implementations and experimental conditions usually differ in many aspects, starting with the hardware used. Therefore, we provide a comparison of factors known to have an important impact in the performance of intrusion-tolerant systems in fault-free executions (see Table 1). The first factor is the number of asynchronous communication steps (*steps* in the table), which has an important influence in the latency of the system since it measures the number of times the communication delay will happen. The second factor is the number of messages sent, which has an impact mostly in the throughput of the system, although also on the latency (*messages* in the table). The third is the number of signatures used, which affects both the latency and throughput of the system because it is a CPU-time consuming operation (*signatures* in the table). The fourth factor is the number of verifications of signatures of the group, usually the verification of signatures of $2f + 1$ or $n - f$ members of the group (*sign. verific.* in the table).

	VIEW INSTALLATION		TOTAL ORDER DELIVERY			
System	Steps	Messages	Steps	Messages	Signatures	Sign.verific.
Rampart / ITUA	6	6(n-1)	3+3	3(n-1)+ 3(n-1)	(n-1)+ (n-1)	(n-1)+ (n-1)
Worm-IT	3	$n(n-1)^{(*)}$	3+3	(n-1)+ $n(n-1)^{(*)}$	0	0
BFT	–	–	5	$2n^2$	0	0

Table 1
Comparison of three intrusion-tolerant systems in fault-free runs.

The table compares the performance of Rampart/ITUA and Worm-IT. There is also a line with data for BFT, which is not a full group communication system, since

its groups are static [12], but has a total order primitive and is a very well-known intrusion-tolerant system in the literature (Castro and Liskov, 2002). The factors related to total order delivery, both for Rampart/ITUA and Worm-IT, are presented as a sum of two values. These two values correspond to the two steps used to deliver a message in total order: (1) to deliver the message to the processors; and (2) to deliver (Rampart/ITUA) or agree (Worm-IT) about the order of the messages delivered. This second cost is not relevant when the throughput is high because there is one ordering step for many messages. We mark with $^{(*)}$ the number of messages sent by Worm-IT to indicate that these do not include the messages sent inside the TTCB. The reason is that in the current implementation each local TTCB periodically sends messages to all others, even if there is no information to send (Correia et al., 2002b), something that also happens in some failure detectors.

The table shows that the performance of the systems can be better of worse depending on the relative impact of the factors. Worm-IT has low numbers of steps; the number of messages is $O(n^2)$ but is low if n is also low. Signatures have been shown to have a high cost in terms of latency and throughput in the experiences made with Rampart, albeit using cryptographic hardware would reduce this cost (Reiter, 1994). BFT is known to have an excellent performance, although the table shows that Worm-IT can probably have a better throughput if the number of messages being sent is high, thus reducing the cost of ordering.

Notice that the table does not say anything about the cost of detecting the failure of the leader/primary, a problem that is shared by both Rampart and BFT. The failure of a process in Worm-IT has no impact per se in the performance, unless the failure means flooding the network with messages, something that would equally affect the other systems.

8 Related Work

The concept of virtually-synchronous group communication (now usually called *view synchrony*) was introduced by Birman and Joseph (1987b,a). The idea was to mimic to some extent a synchronous environment that would deliver all messages and view changes in the same order. However, ordering all messages and events was costly. Therefore, the idea of virtual synchrony was to preserve the illusion of synchrony, but communication primitives with weaker ordering – FIFO, causal, no order – were provided for applications that were insensitive to that aspect. Several variants of this semantics were defined, e.g., extended virtual synchrony (Moser et al., 1994) and weak and strong virtual synchrony (Friedman and van Renesse, 1996).

[12] Therefore the table has no information about view changes for BFT.

Most group communication systems in the literature consider a crash fault model. A recent survey compares many of those systems (Chockler et al., 2001). Some of them have evolved to support a stronger model where the network might be attacked by malicious hackers, but the processors are simply assumed to be secure. Examples of such systems are Ensemble (Rodeh et al., 2001) and Secure Spread (Amir et al., 2005). All these systems have a membership service and view-synchronous communication primitives.

More recently, interest emerged in the problem of designing group communication systems for environments that might suffer arbitrary faults, often called Byzantine faults. We are aware of only three intrusion-tolerant membership services: Rampart (Reiter, 1994, 1996), SecureRing (Kihlstrom et al., 2001, 2003) and SecureGroup (Moser et al., 2000; Moser and Melliar-Smith, 1999). Project ITUA implemented an enhanced version of Rampart (Ramasamy et al., 2002).

The Rampart toolkit provides primitives for reliable (no order) and atomic multicast (Reiter, 1994). Both protocols use digital signatures based on public-key cryptography to sign the messages, which is their performance bottleneck. The atomic multicast protocol relies on an elected sequencer process to order the messages. The failure of this process has to be detected using timeouts. This approach has the problem that slow communication, either due to an attack or to network congestion, can cause correct sequencers to be removed, and if sequencers are repeatedly removed the system does not make progress. The Rampart membership uses a three-phase commit style protocol. Processes in the group send failure suspicions to a leader that tries to change the membership when the majority is received. The sender uses digital signatures to prove that it received the suspicions. The protocol relies on the failure detector to remove a failed leader and make progress, e.g., to eventually install a new view.

SecureRing is designed for LANs and is based on a logical ring imposed on the communication medium. The membership protocol reconfigures the system when one or more processors exhibit detectable Byzantine failures, which are detected by a Byzantine failure detector. Therefore, it suffers from the same failure detection problem as Rampart. The ring has a digitally signed token that passes from process to process. The total order of the communication is implicit in the scheme, since only the process with the token can multicast. The token is used to send checksums of the messages, so normal messages do not have to be signed using public-key cryptography, with benefits in terms of throughput. The failure detector is used to detect if a malicious process does not pass the token or corrupts its information in some way. Token-based protocols have a well-known problem of latency since it augments with the number of processors.

SecureGroup is also designed for LANs (Moser et al., 2000; Moser and Melliar-Smith, 1999). It uses a combination of positive and negative acknowledgments to order messages in causal order using randomization to circumvent the FLP result.

The total order is built on top of this causal order. On the contrary to Rampart and SecureRing, SecureGroup does not provide "multicast" but "broadcast" primitives, i.e., all communication in the domain is ordered. Therefore, it resists a number of intrusions (f) in the processes in the domain, not in the processes in the group, thus its resilience is lower than the resilience of the other systems. The membership protocol is simpler than those of Rampart and SecureRing because it is implemented on the top of the atomic broadcast protocol.

CoBFIT is a framework defined with the objective of supporting the construction of intrusion-tolerant applications using components (Ramasamy et al., 2004). CoBFIT is a framework, not a system, but its usage has been demonstrated using ITUA's intrusion-tolerant group communication system. CoBFIT allows components to be changed in runtime, thus allowing a system to adapt itself when failures happen, usually by replacing a failed component by a new one.

BFT is an algorithm designed with the objective of supporting the implementation of efficient intrusion-tolerant services based on the state machine approach (Castro and Liskov, 2002). A service is implemented by a set of servers that run the clients' requests. BFT is not a full-fledged group communication system since it does not have a membership service and does not provide generic group communication primitives. Nevertheless, BFT totally orders the messages, like Worm-IT. The ordering algorithm is based on a primary processor that defines the order and whose failure has to be detected in order to ensure the progress of the system. However, if the primary is suspected to be failed it is not removed (there are no groups), simply a new primary is selected. This feature is important because it prevents the problem of Rampart and SecureRing, whose safety properties can be impaired by forcing the removal of correct processes from the group simply by delaying and causing a wrong detection.

Recently, Martin and Alvisi proposed an intrusion-tolerant consensus protocol and have shown how it can be used to implement state machine replication (Martin and Alvisi, 2005). The protocol is similar to BFT in the sense that it has also a primary, whose failure has also to be detected to ensure progress. The protocol is said to be fast because it needs only two communication steps to deliver messages in total order when there are no processor failures.

SINTRA is a framework aimed to support the implementation of replicated intrusion-tolerant services (Cachin and Poritz, 2002). It provides a number of group communication primitives, like reliable, atomic and causal multicast. The group model is static, i.e., there is no membership module. The agreement is made using a randomized protocol that relies heavily on public-key cryptography.

Several intrusion-tolerant systems have been designed using Byzantine quorum systems, starting with (Malkhi and Reiter, 1998). COCA is a quorum-based on-line certification-authority for local and wide-area networks (Zhou et al., 2002). COCA

uses replicated servers for availability and intrusion-tolerance. The certificates that it produces are signed using a threshold cryptography algorithm. COCA assumes that tan adversary takes a certain time to corrupt a number of servers, therefore from time to time keys are changed (proactive recovery). Systems based on quorums do not need to make consensus so they are not bounded by the FLP result.

9 Conclusion

This paper explores a novel system architecture and model. Most of the system is asynchronous and susceptible to arbitrary faults, including attacks and intrusions, but it includes a distributed trusted and real-time subsystem called TTCB. This subsystem is an example of a wormhole, a privileged component that provides limited but useful services for applications and protocols otherwise executed in the normal weak environment. The primary contribution of the paper is the presentation of a reasonably complex system that explores this novel architecture. Previous protocols based on the model were considerably simpler (Correia et al., 2002a; Veríssimo et al., 2003; Correia et al., 2005).

Worm-IT is composed of a membership service and a view-synchronous atomic multicast protocol. The system does not suffer from having to detect failures to make progress, a problem present in other systems in the literature. The resilience to intrusions in group members is optimal.

References

Adelsbach, A., Alessandri, D., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J. C., Powell, D., Randell, B., Riordan, J., Ryan, P., Simmonds, W., Stroud, R., Veríssimo, P., Waidner, M., Wespi, A., Jan. 2002. Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21.

Amir, Y., Nita-Rotaru, C., Stanton, J., Tsudik, G., Sep. 2005. Secure Spread: An integrated architecture for secure group communication. IEEE Transactions on Dependable and Secure Computing 2 (3), 248–261.

Avizienis, A., Dec. 1985. The N-version approach to fault tolerant software. IEEE Transactions on Software Engineering 11 (12), 1491–1501.

Baldoni, R., Helary, J., Raynal, M., Tanguy, L., 2003. Consensus in Byzantine asynchronous systems. Journal of Discrete Algorithms 1 (2), 185–210.

Birman, K., Joseph, T., Nov. 1987a. Exploiting virtual synchrony in distributed systems. In: Proceedings of the 11th Symposium on Operating System Principles. pp. 123–138.

Birman, K., Joseph, T., Feb. 1987b. Reliable communication in the presence of failures. ACM Transactions on Computer Systems 5 (1), 46–76.

Birman, K. P., 1997. Building Secure and Reliable Network Applications. Manning Publishing Company and Prentice Hall.

Cachin, C., Poritz, J. A., Jun. 2002. Secure intrusion-tolerant replication on the Internet. In: Proceedings of the International Conference on Dependable Systems and Networks. pp. 167–176.

Casimiro, A., Martins, P., Veríssimo, P., Sep. 2000. How to build a Timely Computing Base using Real-Time Linux. In: Proceedings of the IEEE International Workshop on Factory Communication Systems. pp. 127–134.

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

Castro, M., Rodrigues, R., Liskov, B., Aug. 2003. BASE: Using abstraction to improve fault tolerance. ACM Transactions Computer Systems 21 (3), 236–269.

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

Chockler, G., Keidar, I., Vitenberg, R., Dec. 2001. Group communication specifications: A comprehensive study. ACM Computing Surveys 33 (4), 427–469.

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

Correia, M., Neves, N. F., Lung, L. C., Veríssimo, P., 2005. Low complexity Byzantine-resilient consensus. Distributed Computing 17 (3), 237–249.

Correia, M., Neves, N. F., Veríssimo, P., Jan 2006. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. Computer Journal 41 (1), 82–96.

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

Deswarte, Y., Kanoun, K., Laprie, J. C., 1998. Diversity against accidental and deliberate faults. In: Computer Security, Dependability, & Assurance: From Needs to Solutions. IEEE Press.

Doudou, A., Garbinato, B., Guerraoui, R., May 2002. Encapsulating failure detection: From crash-stop to Byzantine failures. In: International Conference on Reliable Software Technologies. pp. 24–50.

Dwork, C., Lynch, N., Stockmeyer, L., Apr. 1988. Consensus in the presence of partial synchrony. Journal of the ACM 35 (2), 288–323.

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

Fraga, J. S., Powell, D., Aug. 1985. A fault- and intrusion-tolerant file system. In: Proceedings of the 3rd International Conference on Computer Security. pp. 203–218.

Friedman, R., van Renesse, R., Oct. 1996. Strong and weak virtual synchrony in Horus. In: Proceedings of the 15th Symposium on Reliable Distributed Systems. pp. 140–149.

Frier, A., Karlton, P., Kocher, P., Nov. 1996. The SSL 3.0 protocol. Netscape Communications Corp.

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

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

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

Malkhi, D., Reiter, M., Jun. 1997. Unreliable intrusion detection in distributed computations. In: Proceedings of the 10th Computer Security Foundations Workshop. pp. 116–124.

Malkhi, D., Reiter, M., 1998. Byzantine quorum systems. Distributed Computing 11, 203–213.

Martin, J. P., Alvisi, L., Jun. 2005. Fast Byzantine consensus. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks. pp. 402–411.

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

Moser, L., Amir, Y., Melliar-Smith, P., Agarwal, D., Jun. 1994. Extended virtual synchrony. In: The 14th IEEE International Conference on Distributed Computing Systems. pp. 56–65.

Moser, L. E., Melliar-Smith, P. M., 1999. Byzantine-resistant total ordering algorithms. Information and Computation 150, 75–111.

Moser, L. E., Melliar-Smith, P. M., Narasimhan, N., Jan. 2000. The SecureGroup communication system. In: Proceedings of the IEEE Information Survivability Conference. pp. 507–516.

Ramasamy, H., Agbaria, A., Sanders, W. H., Aug. 2004. CoBFIT: A component-based framework for intrusion tolerance. In: Proceedings of the 30th Euromicro Conference. pp. 591–600.

Ramasamy, H., Pandey, P., Lyons, J., Cukier, M., Sanders, W. H., Jun. 2002. Quantifying the cost of providing intrusion tolerance in group communication systems. In: Proceedings of the International Conference on Dependable Systems and Networks. pp. 229–238.

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

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

Rodeh, O., Birman, K., Dolev, D., Aug. 2001. The architecture and performance of security protocols in the Ensemble group communication system. ACM Transactions on Information and System Security 4 (3), 289–319.

Smith, S., 2004. Magic boxes and boots: Security in hardware. IEEE Computer 37 (10), 106–109.

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

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

Veríssimo, P., Neves, N. F., Correia, M., 2003. Intrusion-tolerant architectures: Concepts and design. In: Lemos, R., Gacek, C., Romanovsky, A. (Eds.), Architecting Dependable Systems. Vol. 2677 of Lecture Notes in Computer Science. Springer-Verlag, pp. 3–36.

Zhou, L., Schneider, F., van Renesse, R., Nov. 2002. COCA: A secure distributed on-line certification authority. ACM Transactions on Computer Systems 20 (4), 329–368.

A Correctness Proofs

A.1 Membership Service

This section proves that the membership service satisfies the properties of MS1 Uniqueness, MS2 Validity, MS3 Integrity and MS4 Liveness (Section 5). Here we consider the system model presented in Sections 2 and 3. In these proofs we use f^n to indicate the maximum number of processors allowed to fail in the view number n: $f^n = \lfloor \frac{|V^n|-1}{3} \rfloor$.

Lemma A.1 *For all correct processors P_j that installed V_j^n and V_j^{n+1}, $V_j^{n+1} = V^{n+1}$.*

Proof: Correct processors in view V^n execute the PICK protocol (Alg. 2) to agree on the view changes to V^n that give the new view V^{n+1}. The *last TBA* of the PICK protocol decides the hash of the view changes to be applied to V^n. The proof can be divided in two cases:

Case 1 If at processor P_j the hash decided in the last TBA is equal to *Hash(bag-decisions-tba)* (Alg. 2, lines 8-21), then P_j obtains V^{n+1} by applying the changes in *bag-decisions-tba* to V^n. Therefore, P_j installs V^{n+1}.

Case 2 If that is not true, at least $2f^n+1$ processors proposed for the last TBA (line 19) therefore at least $f^n + 1$ *correct* processors did it (since there are at most f^n failed processors). All correct processors will eventually multicast a PICKED message (line 22), P_j will eventually receive one of them (line 26), terminate PICK (lines 28-29) and install V^{n+1}.

□

Theorem A.2 *If views V_i^n and V_j^n are defined, and processors P_i and P_j are correct, then $V_i^n = V_j^n$ (MS1 Uniqueness).*

Proof: The proof is by induction on views. The group is created by processor P_1 so the initial case is when processor P_2 joins (view V^2). We assume P_2 manages to get a reliable copy of V_1 that contains only P_1 (Section 5.5). Then P_2 sends P_1 a REQ_JOIN and waits for $f^1 + 1 = \frac{|V^1|-1}{3} + 1 = 1$ message with the new view information. Since there is only one processor in V^1, no processor is allowed to fail ($f^1 = 0$), therefore considering the communication model, P_2 gets a correct copy of V^2 and $V_1^2 = V_2^2$.

The proof that $(V_i^n = V_j^n) \Rightarrow (V_i^{n+1} = V_j^{n+1})$ comes directly from Lemma A.1.

Considering that processors can join the group, we have also to prove that $V_i^{n+1} = V_j^{n+1}$ even if V_i^n was defined but V_j^n was not. We assume P_j manages to get a reliable copy of V^n. Then, P_j multicasts a REQ_JOIN and waits for f^n+1 messages with the new view information. Considering the communication model and that there are at least $2f^n+1$ correct processors (at most f^n fail), P_j eventually receives $f^n + 1$ identical copies of V^{n+1} and installs that view. □

Theorem A.3 *If processor P_i is correct and view V_i^n is defined, then $P_i \in V_i^n$ and, for all correct processors $P_j \in V_i^n$, V_j^n is eventually defined (MS2 Validity).*

Proof: The first part of the proof – that $P_i \in V_i^n$ – is trivial. Let us prove that for all correct processors $P_j \in V_i^n$, V_j^n is eventually defined. The case of the first view, V^1, is also trivial. For all other views, V^n is installed after the processors in view V^{n-1} executed the PICK protocol. Therefore, all processors P_j belong to V_i^n for one of two reasons: $P_j \in V^{n-1}$ and P_j did not exit the group; or P_j joined the group to view V^n.

Case 1 $P_j \in V^{n-1}$ and P_j did not exit the group. Let us prove that all correct processors in view V^{n-1} received at least $2f^{n-1} + 1$ messages *(INFO, *, $Ev(P_k)$, *)* from different processors in that view. If V_i^n is defined, PICK terminated in V_i^{n-1}, therefore at least $2f^{n-1} + 1$ processors proposed for the last TBA (Alg. 2, line 19), so at least $f^{n-1} + 1$ *correct* processors proposed for that TBA. For those correct processors to participate in the TBA, they must have received at least $2f^{n-1} + 1$ INFO messages about one event (Alg. 1, lines 17-21). At least $f^{n-1}+1$ of those messages were sent by correct processors so all correct processors will eventually receive them (communication model). A correct processor eventually multicasts an *(INFO, *, $Ev(P_k)$, *)*, either because it "saw" the event or when it receives the $(f^{n-1} + 1)^{th}$ message. Therefore, all correct processors eventually multicast the message and receive it at least $2f^{n-1} + 1$ times, the minimum number of correct processors.

PICK is called with the smallest *valid-tstart* received in the INFO messages (Alg. 1, line 21). Let us now prove that the smallest *valid-tstart* received in any subset of these $2f^{n-1}+1$ messages, from different processors and for this view, is smaller than or equal to the last TBA *tstart* (Alg. 2, line 12). A correct processor multicasts all INFO messages for a view with the same *valid-tstart* (Alg. 1, lines

4, 8-9, 14-16). In any subset of $2f^{n-1}+1$ messages from different processors, at least $f^{n-1}+1$ where sent by correct processors. The last TBA happens when at least $2f^{n-1}+1$ processors manage to propose before *tstart* (Alg. 2, line 19), from which at least $f^{n-1}+1$ are correct. Since there are at least $2f^{n-1}+1$ correct processors, the intersection of these two sets of $f^{n-1}+1$ correct processors has at least one processor. A correct processor cannot propose with a *tstart* smaller than the *valid-tstart* it sends in its INFO messages for a view (Alg. 1, lines 8-10, 14-16). Therefore, the smallest *valid-tstart* a correct processor receives in any subset of $2f^{n-1}+1$ INFO messages is smaller than or equal to the first TBA when $2f^{n-1}+1$ processors manage to propose, i.e., the last TBA.

A processor can receive INFO messages with several events $Ev(P_j)$ in the same view. When it receives for the first time the $(2f^{n-1}+1)^{th}$ INFO message (from different processors) with the same event, it starts executing PICK with the smallest *valid-tstart* in these messages. It calls TBA once or more times, until it gets the result of the last TBA (Alg. 2, lines 8-19) and installs the view, i.e., V_j^n is defined.

Case 2 P_j joined the group. All correct processors in view V^{n-1} eventually install V^n since they are included in Case 1. Therefore, they eventually send the new view information to all processors accepted to join (Alg. 1, line 28). When a processor allowed to join receives $f^{n-1}+1$ of these messages it installs the new view.

□

Lemma A.4 *If any correct processor P_i receives $2f^n+1$ (INFO, *, $Ev(P_j)$, *) messages for the same view V^n and from different senders in the view, then at least one correct processor in the view "saw" the event $Ev(P_j)$. The meaning of "saw" depends on the event: P_i detected the failure of P_j (event* Remv(P_j)*); P_i received a (LEAVE, P_j) message from P_j (event* Leave(P_j)*); or P_i received a (REQ_JOIN, P_j, *) message from P_j (event* Join(P_j)*).*

Proof: In the proof we simply call *message* to a message *(INFO, *, $Ev(P_j)$, *)* and *n messages* to *n* of these messages received from different processors and for the view being considered.

A correct processor multicasts a message for one of two reasons (Alg. 1, respectively lines 6-10 and 11-16) because it "saw" the event $Ev(P_j)$ or because it received f^n+1 messages. Assume no correct processor "saw" the event $Ev(P_j)$. If there are messages it is because a malicious processor has sent it. Since there are at most f^n failed processors, no correct processor receives f^n+1 of messages, therefore no correct processor sends messages, so no correct processor receives $2f^n+1$ messages. □

Theorem A.5 *If processor $P_i \in V_i^n$ and V_i^{n+1} is not defined then at least one correct processor detected that P_i failed or P_i requested to leave. If processor $P_i \in$*

V_i^{n+1} and V_i^n was not defined at P_i then at least one correct processor authorized P_i to join (MS3 Integrity).

Proof: First let us prove the first sentence. If processor $P_i \in V_i^n$ but V_i^{n+1} is never defined then P_i exited the group. This is only possible if PICK made agreement on a *bag-decisions-tba* with an event *Leave(P_i)* or *Remv(P_i)*. A correct processor puts an event $Ev(P_j)$ in *bag-decisions* when it receives the $(2f+1)^{th}$ message *(INFO, *, $Ev(P_j)$, *)* from different processors and for this view (Alg. 1, lines 17-18). Given Lemma A.4, that happens only if at least one correct processor "sees" the event, *Leave(P_i)* or *Remv(P_i)*. To "see" the *Leave(P_i)* means to receive a message *(LEAVE, P_i)* from P_i. To "see" the event *Remv(P_i)* means to detect the failure of P_i. This proves the first sentence. The proof of the second is similar so we skip it for brevity. □

Theorem A.6 *If $\lfloor \frac{|V^n|-1}{3} \rfloor + 1$ correct processors detect that P_i failed or receive a request to join, or one correct processor requests to leave, then eventually V^{n+1} is installed, or the join is rejected (MS4 Liveness).*

Proof: We prove the assertion for the processor failure and skip the proofs for join and leave since they are similar. If $f^n + 1 = \lfloor \frac{|V^n|-1}{3} \rfloor + 1$ correct processors detect that P_i failed they multicast a total of $f^n + 1$ messages *(INFO, *, Remv(P_i))*. There are at least $2f^n + 1$ correct processors and all multicast these messages either because they detected the failure or because they received the mentioned $f^n + 1$ messages. Therefore, all correct processors receive at least $2f^n + 1$ of these messages. If PICK is not running then it starts and eventually decides that view change, considering the weak synchrony assumption made in Section 3. If PICK is already running that change can be decided or not. If not, we assumed the event is re-issued in the next view and that it will be eventually agreed (Section 5.2). □

A.2 VSAM

This section sketches proofs of the correctness of the view-synchronous atomic multicast protocol (VSAM). We consider the system model and the assumptions in Sections 2 and 3.

Theorem A.7 *If a correct processor multicasts a message M, then some correct processor in* group(M) *eventually delivers M (VSAM1 Validity).*

Proof: A processor multicasts a message by calling *VSAM-multicast* (Alg. 3, line 4). The proof that RCAST eventually delivers M to all correct processors is straightforward. Therefore, we know that eventually all correct processors see an event *Datamsg*. The proof that this event is eventually decided by the PICK protocol and the message delivered is similar to the proof of Theorem A.6. □

Theorem A.8 *If a correct processor delivers a message M, then all correct processors in* group(M) *eventually deliver M (VSAM2 Agreement).*

Proof: If a correct processor delivers M then it previously resends M to all processes that it is not aware to have the message (Alg. 3, line 24). This guarantees that all correct processes eventually *receive* M. The proof that all processes eventually *deliver* M follows from Theorem A.6. □

Theorem A.9 *For any message M, every correct processor 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) *(VSAM3 Integrity).*

Proof: The proof follows from the secure channels model (Section 3.1). □

Theorem A.10 *If two correct processors deliver two messages M_1 and M_2 then both processors deliver the two messages in the same order (VSAM4 Total order).*

Proof: If M_1 and M_2 are delivered in consequence of the result of the same PICK execution the proof is obvious (see Algorithm 1, line 21). The proof that a correct processor eventually engages in an execution of PICK follows from Theorem A.6. This has also the consequence that all correct processors execute the same sequence of PICKs in a view. Therefore, if M_1 and M_2 are delivered in two different executions of PICK, the property is also satisfied. □

Theorem A.11 *If two correct processors install views V^n and V^{n+1} then both processors deliver the same messages in view V^n (VSAM5 View synchrony).*

Proof: Any correct processor delivers a VSAM message only if the PICK protocol says so. New views are also installed after the execution of a PICK protocol, which is similar to PICK, therefore the proof follows from the proof of Theorem A.2. □