Finite Memory: a Vulnerability of Intrusion-Tolerant Systems *

Giuliana Santos Veronese[1] Miguel Correia[1] Lau Cheuk Lung[2] Paulo Verissimo[1]
[1]LASIGE, Faculdade de Ciências da Universidade de Lisboa
[2]Departamento de Informática e Estatística, Centro Tecnológico, Universidade Federal de Santa Catarina
giuliana@lasige.di.fc.ul.pt mpc@di.fc.ul.pt lau.lung@inf.ufsc.br pjv@di.fc.ul.pt

Abstract

In environments like the Internet, faults follow unusual patterns, dictated by the combination of malicious attacks with accidental faults such as long communication delays caused by temporary network partitions. In this scenario, attackers can force buffer overflows in order to leave the system in an inconsistent state or to prevent it from doing progress, causing a denial of service. This paper is about the effects that finite memory has on intrusion-tolerant protocols and systems. We present the problem and propose a generic mitigation technique based on repair nodes that reduces the buffer space requirements. An experimental evaluation of the buffer usage with and without this technique is presented, allowing to assess in practice the effects of finite memory in a real, albeit simple, intrusion-tolerant system.

1 Introduction

Intrusion tolerance has been proposed for some years now as a new paradigm for computer systems security [8, 24, 15]. The idea is to apply the fault tolerance paradigm in the domain of systems security, accepting

that malicious faults (attacks, intrusions) can never be entirely prevented, and that highly resilient systems have to *tolerate* these faults. The main motivation for new security paradigms like intrusion tolerance has been the poor state of security in the Internet, yearly reported in documents like [10].

Research in intrusion tolerance has been centered in designing intrusion-tolerant distributed systems, based on message-passing protocols [4, 5, 6, 7, 13, 15, 19, 22] Intrusion tolerance is usually obtained by replicating the system in a set of servers, which behave according to the system specification even if there are intrusions in up to a certain threshold of the servers. If each server is protected using the current best practices, and there is diversity between the servers in such a way that they do not share the same vulnerabilities [17], the overall system is ensured to be more trustworthy than if it was centralized.

Research in this area has produced a set of clever intrusion-tolerant protocols and systems (*I/T protocols* and *I/T systems* for short). However, we believe that an issue has been overlooked: that servers have *finite memory*, so the number of messages that can be stored in their *buffers* is limited. Intuitively, this can be a problem in systems in which there are many messages being exchanged. Moreover, all of these systems assume that the environment is essentially asynchronous, i.e., that there are no bounds on communication and processing delays. Assuming this kind of model is very important in order to prevent the success of attacks against time.

However, this combination of a limited capacity to store messages with long message delays that cause long protocol execution times, can be very problematic. This is the crucial problem debated in this paper: the effects that finite memory has on I/T protocols and systems. In environments like the Internet, faults follow unusual patterns, dictated by the combination of malicious attacks with natural faults such as long communication delays, e.g., due to temporary network partitions. In this scenario, attackers can force *buffer overflows*[1] in order to leave the system in an inconsistent state or to prevent it from doing progress, causing a denial of service.

The paper starts by presenting the problem and showing that there are three levels at which it is not possible to guarantee that messages are safely removed from the buffers : *channel, protocol instances and service* levels. After presenting the problem, the paper studies buffer overflows with an I/T group communication primitive, inspired by the Rampart toolkit [22]. Our system provides only an echo multicast primitive similar to Rampart's. However, unlike Rampart, our system is not a full-fledged group communication system, since it has no membership service (groups are static). This simplicity of the system is on purpose since it allow us to put the emphasis in the effects of finite memory and buffer management, not in the system design, which would require a full paper on its own. After presenting the multicast primitive, we propose a generic scheme to deal with message losses caused by buffer overflows based on the notion of *repair nodes*. This scheme allows messages to be stored only in some of the system's processes, thus improving the memory usage of the overall system and increasing the possibility of the message being delivered in case the network is experiencing a temporary disconnection. Repair nodes do not solve the

*This work was supported by the EC through project IST-4-027513-STP (CRUTIAL), NoE IST-4-026764-NOE (ReSIST) and Alban scholarship E05D057126BR, by the FCT through the Multiannual and the CMU-Portugal Programmes, and by CAPES/GRICES through project TISD.

[1]These buffer overflows should not be confused with the common C/C++ buffer overflow attacks. These latter attacks consist in injecting data in a buffer for which the limits are not checked, writing over memory used for other purposes, with effects that may range from crashing the application to running arbitrary code on the attacked machine.

buffer overflow problem, but reduce significantly its impact in the system. However, the buffer overflow problem in this kind of system is essentially unsolvable, so mitigating it is the best that can be done. Finally, an experimental evaluation of the buffer usage with and without the repair node scheme is presented, allowing to assess in practice the effects of finite memory in a real, albeit simple, I/T system.

The contributions of the paper are the following:

- the first systematic study of the problem of buffer overflows in intrusion-tolerant systems and protocols;
- a generic scheme based on repair nodes to allow messages to be available for longer time and be recovered in case they are lost;
- a practical assessment of the buffer overflow problem and the repair nodes scheme in a simple intrusion-tolerant system.

2 System Model

We consider a distributed system that contains a group of n processes $\mathscr{P} = \{p_1, p_2, ..., p_n\}$ that communicate only by message-passing. This group of processes is closed, i.e., no processes join or leave the group. The system is *asynchronous*, so there are no bounds on processing or communication delays. For I/T protocols, there is a common understanding that protocols should not rely on time assumptions about communication or processing, at least to guarantee that safety properties are satisfied. The reason is that time assumptions are known to be vulnerable to attacks that delay the system, e.g., by flooding the network with traffic (a denial of service attack, DoS), or causing temporary network disconnections.

The connection between processes is modeled in two ways in the paper. In Section 3 we consider that each pair of processes is connected by an *authenticated fair link*, which can lose messages but that delivers infinite times messages that are sent infinite times [1]. These links can be easily obtained in the internet using UDP over IPsec/AH –Authentication Header protocol [12]. They guarantee the *authentication* of the processes, i.e., that they have certainty about who is the sender, usually using message authentication codes (MACs) obtained using a cryptographic key shared by the two processes and a cryptographic hash function [14]. Details about how to distribute these keys are out of the context of the paper.

For the protocol in Section 4.1, we assume processes are connected by *authenticated reliable point-to-point channels*, which can be (approximately) implemented on top of authenticated fair links, but also guarantee the reliability of the communication, i.e., that messages are all delivered, unmodified, using retransmissions and MACs. These channels can be obtained using TCP over IPsec/AH or SSL/TLS [9]. However, enforcing reliability involves storing messages in a *send buffer* of the sender until it receives an acknowledgment that the message was delivered to the recipient. This buffer must have limited size due to the finite size of the sender's memory, so the sender may face a situation in which this buffer is full (see Section 3.1). We assume the existence of a *cryptographic hash function* $H(m)$. This function is assumed to be *collision-resistant*, i.e., that $\forall m, m' : H(m) = H(m') \Rightarrow m = m'$. The output of this function is called a *hash*.

Processes can be *correct* or *faulty*. Correct processes always follow their protocol. No assumptions are made about the behavior of *faulty* processes. They can collude against the correct processes following some malicious purpose. This class of unconstrained faults is usually called *arbitrary* or *Byzantine*. We follow the literature and sometimes also say that some faulty processes are *malicious* if they have the intention of breaking the behavior of a protocol or system. We assume that at most $f = \lfloor (n-1)/3 \rfloor$ processes can be faulty, which implies that $n \geq 3f + 1$.

3 The Problem

This section presents a systematic study of the problem of buffer overflows [2] in intrusion-tolerant systems and protocols. The main goal is to give a formal description of the problem looking at three different levels: channels, protocol instances and service.

3.1 Channels

Consider a set of processes that communicate by message-passing over *authenticated fair links*, which deliver infinite times messages sent infinite times. The system is asynchronous. Consider also an application that repeatedly requests a process p to send *data messages* (i.e., application-level messages) to a process q for a long period of time. The application wants to be sure that q receives the messages, so messages are numbered sequentially and whenever q receives a message $m(k)$ it has to send a message $ack(k)$ to p, where k is the number of message m. p stores all the messages it sends in a *send buffer*; when p receives $ack(k)$, it discards $m(k)$, since it knows that q received it. Messages that are not confirmed have to be retransmitted after a timeout.

Now, suppose that after a certain instant, p stops receiving acknowledgments from q. If the application periodically requests p to send messages to q, eventually p's *send buffer* will be full of messages (memory is finite). Therefore, when the application requests it to send the next message, say m', it has three possibilities: (1) discard the message m', hoping that q is faulty; (2) discard an older message, hoping that q received it; (3) block waiting for q to acknowledge messages, hoping that the communication with q is slow or there is a temporary disconnection, but that it will recover. The problem is that p can not know for sure if q is faulty or the communication is simply experiencing long delays, so all these solutions are problematic. In (1), if in reality there is a temporary disconnection, q will never receive the message. In (2), if q is correct and did not receive the message, it will never receive it. In (3), if q is faulty, p will stay blocked.

This is the kind of dilemma that asynchronous I/T protocols have to deal with. The problem is that due to the limited size of memory, and consequently of buffers, a protocol may have to sacrifice either a safety property (e.g., discarding messages) or a liveness property (blocking), in both cases poten-

[2] Recall once more that these buffer overflows are not be confused with buffer overflow attacks (see footnote 1)

tially impairing the behavior of the protocol or system. Notice that this problem of the buffer size being limited is *not* an implementation detail, but an intrinsic, theoretical, problem, which can not be solved simply by making this size larger.

Most I/T protocols in the literature assume that processes are fully-connected by *authenticated reliable point-to-point channels* (that deliver all messages), or implement those channels on top of authenticated fair links (e.g., [20, 22]). The problem of reliable channels with finite memory is essentially the problem just described, which can be stated more formally as:

Proposition 1 *It is not possible to implement an authenticated reliable point-to-point channel on top of authenticated fair links that eventually relinquishes messages from the send buffer (i.e., frees buffer memory) if the processes can fail in a Byzantine way.*

3.2 Protocol Instances

I/T systems often run several instances of the same I/T protocol(s) concurrently. In [20, 22] several processes can be sending messages in parallel to the others, using several communication primitives. Malicious processes can send messages about a non-existent instance of a protocol, which the others have to store and can not discard because they can not distinguish an instance that does not exist from one that they are not aware of. The problem can be stated:

Proposition 2 *If for a certain I/T protocol, the relinquishing of messages from the internal buffers involves receiving the message plus some form of confirmation from at least a process other than the sender, and there can be an arbitrary number of parallel executions of instances of that protocol, then it is not possible to guarantee both the safety and the termination of all instances of the protocol.*

3.3 Service

Most I/T systems in the literature replicate a *service* in a set of servers that is accessed by a set of clients. This is the case for the two techniques most used to implement I/T systems: state machine replication [5] and Byzantine quorum systems [19]. Clients send requests to the servers, which process the requests and send back replies.

The buffer problem at *service level* is the following. The servers have to store in a *reply buffer* the replies that they have to send to the clients. If clients do not acknowledge the reception of the replies, either because they are faulty or because the network is experiencing long delays or disconnections, this buffer can overflow. In those cases, servers either stop accepting more requests –blocking the service– or discard replies –possibly leaving correct but slow clients without the replies they are entitled to. The problem at *service level* can be stated as:

Proposition 3 *It is not possible to implement a reliable service that eventually relinquishes reply messages if the clients can fail in Byzantine way and the links are fair.*

4 Buffer Overflows in an I/T System

After presenting the problem, we now study buffer overflows with an I/T group communication primitive, inspired by the Rampart toolkit [22] and RITAS [20]. The objective is not to present a full-fledged I/T system that solves the buffer overflow problem, which would require a paper on its own. In contrast, the idea is to show that the problem can have a relevant impact in a real, albeit simple, system, and to present a generic mitigation mechanism that can be implemented in any I/T System. The mechanism is generic in the sense that it can be used in many different I/T systems.

4.1 Echo Multicast

In order to study the buffer overflow problem in I/T protocols, we implement a *message dissemination protocol* based on the *echo broadcast* proposed by Toueg [23], later modified by Reiter for Rampart [22]. Those protocols guarantee two properties: (1) if a correct process sends a *data message* (or application-level message), all correct processes deliver that message; and (2) no two correct processes deliver two different data messages with the same identifier (if the sender is malicious, some of the correct processes may not deliver the message, but all that do, deliver the same).

We assume the system model in Section 2, with authenticated reliable point-to-point channels. The protocol, presented in Algorithm 1, satisfies the properties (1) and (2) above if the channels have unlimited capacity (i.e., if the send buffer is infinite), so we assume infinite memory in this section. However, in practice memory is finite, so in the next section we start assuming limited-capacity authenticated reliable point-to-point channels.

In the original Toueg's protocol, when the first message is received, all correct processes reply with a message containing the same data. We propose an optimization in which each process replies with a message containing only the hash of this data. This approach reduces the use of buffer space.

Messages are composed by fields such as: ⟨PROTOCOL, MESSAGE TYPE, SENDER IDENTIFIER, MESSAGE IDENTIFIER, DATA, HASH⟩. All messages that are sent by a process p_i are tagged with a message identifier (or sequence number) eliminating possible interferences among multicasts. There are two types of messages: *initial* and *echo*. The protocol starts with the sender transmitting an *initial* message m. All processes that received m send to each other an *echo* message containing the *hash* of m. If a process receives an *initial* and $(n-f)$ *echo* messages with the same *hash*, then it accepts m.

Algorithm 1 Message dissemination protocol

when p_i wants to send a data message m **do**
  send ⟨MULTICAST, initial, p_i, $seq(m)$, m⟩ to all processes

when p_j receives from p_i a message ⟨MULTICAST, initial, p_i, $seq(m)$, m⟩ **do**
  send ⟨MULTICAST, echo, p_j, $seq(m)$, $H(m)$⟩ to all processes

when p_j receives messages ⟨MULTICAST, echo ,*, $seq(m)$, $H(m)$⟩ from $(n-f)$ distinct processes (and p_j receives from p_i a message ⟨MULTICAST, initial, p_i,$seq(m)$, m⟩) **do**
  Accept m

The protocol is presented in Algorithm 1. In the second *when*, any subsequent ⟨MULTICAST, INITIAL, p_i, $seq(m)$, *, *⟩ message received by p_j from the same process p_i is

ignored. A proof that the modified protocol satisfies the same properties as Toueg's original protocol that is straightforward so we do not include it in the paper.

4.2 Buffer Management

Suppose we have a system, like Rampart, in which processes communicate using the echo multicast protocol in the previous section. There are several I/T systems that use similar communication primitives to do exactly that. However, in practice echo multicast must be implemented on top of limited-capacity authenticated reliable point-to-point channels since memory is finite, and in Section 3 we showed that it is impossible to guarantee that these channels will free their *send buffers*. In the same section we also showed that with an arbitrary number of parallel executions of instances of a protocol like the echo multicast, the system can deadlock due to a buffer overflow.

These problems force processes in certain circumstances to choose either (1) to block or (2) to discard messages. Both options are highly undesirable but we believe (1) to be the worse because the system stops (or may stop) functioning. Moreover, although it is usually not reported, (2) is in general the solution used, as confirmed by our own experience and by colleagues that implemented I/T systems. Nevertheless, these systems are carefully configured to avoid having to discard messages under normal operation, and even under attacks not specifically targeted to cause buffer overflows.

If messages are discarded it is still a good idea to try to store them in some of the processes and retransmit them if some processes lose some of them. This would allow the system to recover, e.g., from temporary network partitions that cause the unavailability of one or more processes. Another solution would be for the processes that received all messages to do a state transfer to the process(es) that lost the system state. However, this is a costly operation and sometimes it is not even possible if all processes lose some messages.

Therefore, the problem is to determine which processes should buffer a message and how long they should keep it. The expected approach would be to try to store the messages in all processes that have them for as long as possible. However, this is clearly a bad option because the total memory available in the system is limited, so we should avoid occupying it with several copies of the same data.

In the next sections we present a solution to optimize the buffer management in I/T protocols. Buffer overflows at *channel level* are "solved" by discarding messages that can not be put in the *send buffer* (in fact, in the experiments we try to send them twice with a small time interval between tries). We deal essentially with the problem of buffer overflow at *protocol instances*. The solution is based on discarding messages to avoid that the system blocks. However, the system uses *repair nodes* (or repair processes) to retain messages in the system as long as possible, increasing the possibility of recovery and decreasing the use of state transfers if some processes lose messages. We do not consider the problem at *service level*, since we do not have a service, only a communication primitive, but repair nodes might also be used easily at this level.

Message Storage. This section presents our strategy to store messages received by processes using the dissemination protocol described in Section 4.1.

When a process receives the first message of a protocol instance, it creates a *context object* [3] in order to store information about the messages received for that instance (*initial* and *echo* messages). All contexts are stored in a *context buffer*. Contexts are identified by the sender identifier and the *initial* message identifier, and they are classified in states: *progress*, *accepted* and *end*. Contexts with less than $n-f$ messages received from different processes (or without *initial* message) are in the *progress* state. Contexts with one *initial* message and at least $n-f$ *echo* messages (but less than n) are in the *accepted* state. Contexts with *initial* message and n *echo* messages are in the *end* state. Accepted contexts only remain in the *context buffer* of *repair nodes*. Contexts in the *end* state are removed from all *context buffers* (all processes got the data message). When a process that is not a repair node decides discard a context that is not int the *end* state, it calculates the hash of the *initial* message and before it discards the context, it stores this hash in another buffer, called the *hash buffer*. The hash is identified by the sender and the message identifiers and it will be used in the message recovery protocol that is presented bellow.

A process is defined to be a repair node for a certain data message depending on the corresponding *initial* message identifier. A global system parameter, N_{rn}, defines how many processes are repair nodes for each message. This parameter should be defined at least as $N_{rn} = f+1$ to guarantee that at least one correct node will store each message. However, since any process can lose a message, choosing a higher value for N_{rn} increases the possibility of a message being available, but also involves more nodes storing the message, so a worse buffer usage. The repair nodes for a message with identifier *id* are obtained by calculating *id modulo* $\binom{n}{N_{rn}}$ and using this value as an index of a vector V with subsets of the processes. An example of how the algorithm works considering $N_{rn} = f+1$ and $n = 4$ is:

$$n = 4 \;\rightarrow\; \mathscr{P} = \{1,2,3,4\} \;\rightarrow\; N_{rn} = f+1 = 2$$
$$V[0] = \{1,2\},\; V[1] = \{1,3\},\; V[2] = \{1,4\},$$
$$V[3] = \{2,3\},\; V[4] = \{2,4\},\; V[5] = \{3,4\}$$

Message $id = 983 \;\rightarrow\; 983$ *modulo* $6 = 5 \;\rightarrow$ repair nodes = {3,4}

Using repair nodes allows a better usage of the system memory. The benefit depends on N_{rn}. If $N_{rn} = f+1$ and $n = 3f+1$, the gain of extra space available is $\frac{3f+1}{f+1}$, i.e., from 2 (for $f = 1$) to 3 (when $f \rightarrow \infty$); if $N_{rn} = f+1$ and $n \gg f$ the gain tends to n/f. Summarizing, message storage can be considered to be done at two time-scales:

Short-term: when an instance of the echo multicast is executed, initially all nodes that received the *initial* message keep it in the *context buffer* until they receive $n-f$ messages and the data is accepted.

Long-term: when an instance of the echo multicast is executed and the data is accepted (but less than n *echos* are received), the message is stored only in the repair nodes, but the other processes store a hash of the message in the *hash buffer*, to assist the recovery procedure (see below).

[3] Or context data structure, but we prefer to call it object since the prototype was implemented in an object-oriented language, Java

The general idea is that the responsibility of message buffering is shared by all processes but each one stores only a subset of the messages. This storage approach can be used with any I/T protocol.

Message Recovery. In our system and with our system model, a message can fail to be received by a process mostly for three reasons: a buffer overflow of the *send buffer* of the sender; the sender is malicious and does not send the message on purpose; the message was garbage collected from the *context buffer* (see below). In this section we present a message recovery protocol with the purpose of increasing the chances that a message is eventually received, while avoiding to block the system. Notice that it is impossible to guarantee that no messages are lost at all without blocking the system, since, for instance, we allow disconnections with arbitrarily long duration and with the processes continuing to send messages.

Algorithm 2 Message recovery protocol

when p_i wants to request a missing *echo* message **do**
  send $\langle$RECOVER, echoRequest, p_i, p_z, $seq(m)\rangle$ to all processes from which an *echo* was still not received

when p_j receives from p_i a message $\langle$RECOVER, echoRequest, p_i, p_z, $seq(m)\rangle$ **do**
  if m with sender p_z and message identifier $seq(m)$ is in the *context buffer* **or** $(p_z, seq(m), H(m))$ is in the *hash buffer* **then**
    send $\langle$MULTICAST, echo, p_j, $seq(m)$, $H(m)\rangle$ to process p_i

when p_i wants to request a missing *initial* message **do**
  send $\langle$RECOVER, initialRequest, p_i, p_z, $seq(m)$ $\rangle$ to all processes

when p_j receives from p_i a message $\langle$RECOVER, initialRequest, p_i, p_z, $seq(m)\rangle$ **do**
  if m with sender p_z and message identifier $seq(m)$ is in the *context buffer* **then**
    send $\langle$MULTICAST, initial, p_z, $seq(m)$, $m\rangle$ to process p_i

The message recovery protocol has two parts: one to recover *echo* messages and another to recover *initial* messages (Algorithm 2). It works essentially as follows.

The recovery mechanism periodically analyzes each context in the *context buffer*. If the context has no *initial* message or has less than $n-f$ *echo* messages and remains in the buffer more than T_{recov} units of time, a control message is sent to all other processes requesting the missing message(s). The control message has the format: $\langle$PROTOCOL, MESSAGE TYPE, REQUESTER IDENTIFIER, SENDER IDENTIFIER, MESSAGE IDENTIFIER$\rangle$. All processes that receive a control message reply with the message requested, if they still have it. More precisely, when a process receives an *echoRequest* message and the corresponding context remains in its *context buffer* or the *hash* is in the *hash buffer*, it sends an *echo* message. The *hash* is important because even processes that are not repair nodes participate in the recovery protocol by sending *echo* messages. The idea for *initialRequest* is the same, but the *initial* message is sent only if the context remains in the *context buffer*. The *echo* and *initial* messages delivered by those algorithms are delivered to the dissemination protocol (Algorithm 1). Fake messages sent by malicious repair nodes are discarded by the recipient since they can never match the *initial* message (if they are *echo* messages) or $n-f$ *echos* (if they are *initial* messages).

T_{recov} is an application-specific parameter that gives the maximum time that a context stays in the *context buffer* before the recovery protocol is executed. N_{recov} is another parameter that gives the maximum number of times the recovery protocol is executed for each context. Malicious processes can try to cause buffer overflows by sending messages to less than $2f+1$ correct recipients. In that case there are not enough *echos* and the message can never be accepted. Furthermore malicious processes can forge *echo* messages to make the receivers create contexts for which there is no *initial* message. The N_{recov} parameter serves to avoid that the recovery protocol is executed forever when it is not possible to recover. After N_{recov} recovery executions, if the context is not in the accepted state and has less than n *echo* messages, it is removed.

Garbage Collection. Our system uses two *garbage collectors* to avoid entirely buffer overflows of the *context* and *hash buffers*: the *context collector* and the *age-based collector*.

Consider a process p_i. Messages for which p_i is *not* a repair node are discarded from the *context buffer* by the *context collector* (in the short-term). This collector is executed periodically, with period T_{recov}, and it does the following (only for contexts for which p_i is not a repair node): (1) remove contexts in the *progress* state that remain in the buffer after N_{recov} recovery protocol executions; (2) remove contexts in the *accepted* state; (3) remove contexts in the *end* state.

The idea of using repair nodes is to store information about messages as long as possible. In the long-term, contexts may have to be removed even from the *context buffer* of their repair nodes, to avoid a protocol instance to remain in the buffer forever. This removal is performed by a specific collector that only deletes objects when space is needed, the *age-based collector*. It is executed whenever the *context buffer* free space drops below a low-water mark. The age-based collector simply discards the N_{old} oldest contexts. N_{old} is a parameter specified by application. An age-based collector is also associated to the *hash buffer*, to remove old hashes.

4.3 Evaluation

In order to understand the effects of finite buffers on I/T protocols/systems, we did a set of experiments with a prototype of the echo multicast protocol written in Java, using three different *buffer management policies*:

P1 – no removals: messages are stored by all processes and are discarded only when all processes confirm the message reception. There are no repair nodes and nothing is done to deal with buffer overflows. The system stays blocked waiting for free space in the buffer.

P2 – with repair nodes (our proposal): in this approach the responsibility of message buffering is shared by all processes, however every process stores only a subset of messages in the long-term. Messages are discarded from the buffers by the context and age-based collectors (see Section 4.2).

P3 – without repair nodes: messages are stored by all processes and are discarded only when all processes confirm the message reception (since other processes may

request its retransmission). If the buffer is full, the oldest messages are discarded by the age-based collector.

All the three policies use the recovery protocol presented in Section 4.2. Policies P1 and P3 do not have repair nodes, so they also do not have a *hash buffer*. For policy P3, that means that an *echo* message can only be resent before the process to which it is requested discards the corresponding context (in the case of policy P1, contexts are not discarded). The experiments were run on the Emulab environment [25], on 16 Pentium-III machines with 850 Mhz processors, 512 Mb of RAM and Red-Hat Linux 9. The JVM was Sun JDK1.5.11.

In all tests the virtual machine memory was limited to 100Mbytes to allow the experiments to assess the impact of finite memory without having to run for long periods of time. We also executed some of the experiments with much larger memory and the problems were the same, but the experiments took much longer times. Notice that these protocols are part of the middleware (e.g., analogous to RPCs or CORBA) so they are supposed to leave most of the memory to the service/application, not consume it themselves.

For policy P2 the number of repair nodes was $N_{rn} = f + 1$. For all policies, the maximum number of recovery protocol executions per context was $N_{recov} = 2$ and the time interval between recovery executions was $T_{recov} = 3$ seconds. The experiments were executed with different message sizes: 1, 10 and 100Kbytes. The value of f varied from 1 to 5 and the number of processes was set to $n = 3f + 1$, so varied from 4 to 16.

The measurements were taken under three different faultloads. In the *failure-free faultload*, all processes behave correctly. In the *fail-stop faultload*, f processes crashed before messages started to be sent. In the *Byzantine faultload*, f malicious processes tried to cause buffer overflows in two ways: (1) for each message they had to send, they sent two different messages, one to each half of the recipients, but both with the same identifier, making them impossible to deliver; (2) they never sent *echo* messages, so correct processes did not receive *echos* from all processes and were not able to discard these contexts.

Effect of Buffer Overflows. The objective of the first set of experiments is to show the occurrence of buffer overflows. For that purpose, the system was executed with policy P1, which simply blocks waiting for space when a buffer is full, under a Byzantine faultload. All experiments had exactly the same effect: after some time the execution of the system blocked entirely. The numbers of data messages/contexts in the *context buffer* when the overflows occurred are displayed in Figure 1. The results were similar with different values of n, which was expected since the capacity of the *context buffers* does not change with n. For 100Kbytes messages, the number of messages in the buffer was approximately 850 (line on the bottom).

Time to Discard the First Data Message. The main interest of the repair node scheme we propose is to increase the time a data message is available in the system in case there is a temporary network partition or high communication delays. In this section we begin to evaluate the benefit of having this scheme (policy P2), instead of storing the messages in all processes that receive them (policy P3).

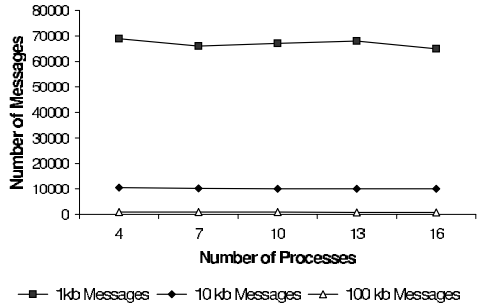

Figure 1. Number of contexts in the *context buffer* when a buffer overflow happens and the system blocks (Byzantine faultload)

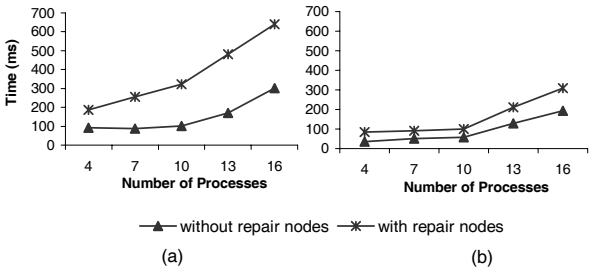

Figure 2. Time to discard the first data message, measured with (a) fail-stop and (b) Byzantine faultloads, with and without repair nodes

This first evaluation consists in measuring the time until the first data message is discarded from a *context buffer* due to a buffer overflow with both policies. The set of experiments presented in Figure 2 evaluated this time with the fail-stop and Byzantine faultloads. Each test was executed 10 times and each process (even if Byzantine) sent $20000/n$ data messages with 10Kbytes at a rate of 100 messages per second.

The number of repair nodes was $N_{rn} = f + 1$, so the results in the figure are the expected: the system with repair buffers discarded the first message at least twice as late as the system without repair nodes. In fact, the ratio between the time to discard in policies P2 and P3 is approximately proportional to N_{rn}/n, as also expected. The time to discard the first message with repair nodes (P2) might be further improved by reducing this ratio, e.g., by keeping $N_{rn} = f + 1$ constant but increasing the total number of processes n. In the Byzantine faultload, data messages are discarded earlier than in the fail-stop faultload, since Byzantine processes do their best to cause discarding.

Percentage of Data Messages Lost. The second part of the evaluation of the repair node scheme consists in measuring the number of data messages/contexts lost due to garbage collections with and without repair nodes. A data message is said to be lost when it is removed from enough processes for not being delivered by all correct processes.

The set of experiments are presented in Figure 3. Each test was executed 10 times and in each test each process sent $20000/n$ data messages. The message size used was 10Kbytes and the maximum number of contexts in the buffer was set to 7000, which is approximately the maximum num-

ber of messages that can be stored with 100Mbytes of memory in the virtual machine [4]. Whenever the number of contexts reached this value, the age-based collector removed the oldest contexts. Each process sent one data message every 30ms. The percentage of messages/contexts lost in the figure is the average of the tests.

The figure shows that the percentage of data messages lost with repair nodes (policy P2) was lower than the number measured without repair nodes (policy P3) for both faultloads and all values of n. These results were expected since using repair nodes, the time a data message is available in the system is larger, so it is possible to recover it in case it is discarded from some buffer. Furthermore, the benefit was higher with Byzantine processes attempting to cause buffer overflows on purpose.

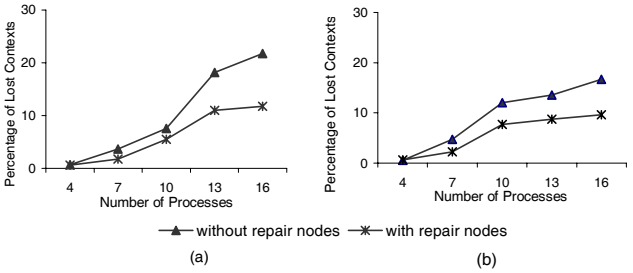

Figure 3. Percentage of data messages lost measured with (a) fail-stop and (b) Byzantine faultload, with and without repair nodes

Buffer Occupation. Figure 4 presents the number of data messages/contexts in the *context buffer* in a set of experiments. The experiments were conducted with 10 processes using policies P2 and P3. Each process sent 1000 messages at a rate of 100 messages per second, with intervals of 1 minute between these bursts of 100 messages. Values in the graphs were taken in a single process. Periodically, with period of 1 second, one thread obtained the number of contexts in the buffer. These are the values presented in the figure.

The figure leads to several interesting conclusions. With the fault-free faultload, when there is a burst the buffer occupation increases, but then decreases fast until the next burst. The buffer occupation tends to be low but there are some message losses (probably *echoes*) that prevent some of the contexts from being removed from the buffer.

With the fail-stop and Byzantine fault-loads, not all *echoes* are received so the buffer goes on being filled up with messages at a constant pace. At some point the buffer becomes full and the age-based collector starts discarding messages to prevent buffer overflows (the stable zone of the lines in Figures (b) and (c)). These two figures confirm that the buffer occupation is better with than without repair nodes, since the buffers are filled slower with repair nodes.

Discussion. Before the experiments reported in this section were done, many tests were performed in order to discover the best and more realistic configuration parameters: the rate at which messages should be sent (or the interval between send events) in order to avoid as much as possible *send buffer* overflows, the number of recovery executions N_{recov} and the period of recovery T_{recov}. The values of the parameters used were the same with all buffer management policies. The size of the *hash buffer* was defined by observing how many hashes were requested by correct processes that executed the recovery protocol, in many experiments with the three faultloads. In all tests the number of requests did not exceed 800 hashes, which is approximately 10% of the *context buffer* size. Therefore, for experiments with policy P2 the size of the *hash buffer* was set to hold 1000 hashes.

The experiments on the effect of buffer overflows have shown that the problem is real and that the processes can block due to lack of memory. These experiments also showed how many messages can be stored with limited memory (with virtual machines with 100Mbytes).

In all experiments we had better results when using the repair nodes scheme, i.e., with policy P2, since the weight of buffering messages in the long term is scattered by all processes, instead of being shared by all for all messages. The time a data message is available in the system is longer, so more messages can be recovered. This belief was supported by the experiments that have shown that less messages were lost when using repair nodes.

5 Related Work

The buffer overflow problem at *channel level* derives from the classical *two generals problem*, which is the problem of two processes connected by fair links reaching agreement on the delivery of a message [2, 11]. The problem is unsolvable if the network can lose messages, even if the processes are correct. Several papers explored this issue of implementing reliable point-to-point and multicast channels with different types of links: fair links, fair-lossy links, eventually reliable links, unreliable links [3, 16, 21]. Several results on the area were surveyed in [18]. The mechanisms used to implement these communication primitives can be extended for Byzantine faults using cryptographic techniques, such as message authentication codes (MACs), which can be used to guarantee the authentication and the integrity of the messages exchanged [14].

Many intrusion-tolerant protocols and systems have been proposed in the literature, including several by some of this paper's authors. However, to the best of our knowledge, the only work that discusses the problem of having finite buffers and garbage collection with some detail is BFT [5]. BFT does not use reliable channels, but authenticated fair links implemented with UDP and MACs, doing retransmissions when needed. This circumvents the problem of buffer overflows at *channel level* but requires the servers to store messages in a buffer called *message log*, analogous to what we call *context buffer*. This buffer can overflow, so BFT has a garbage collection mechanism that discards messages related to requests already executed. Messages are not discarded one at a time for each request executed, but when a *checkpoint* for a number of messages is obtained. Although this mechanism is very effective, several attacks that try to fill this buffer seem

[4] The objective of imposing a maximum number of messages is to avoid having to check if there is enough memory for each message that has to be stored in the buffer, since in Java overflowing the virtual machine memory raises a fatal exception that can not be handled.

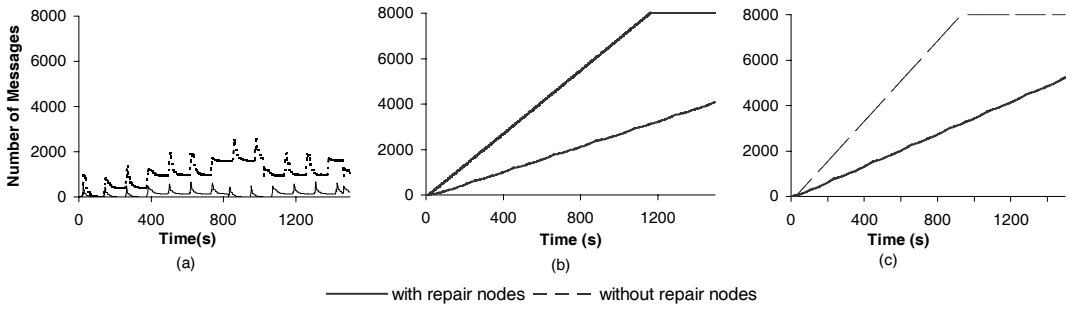

Figure 4. Buffer occupation measured with (a) fault-free, (b) fail-stop and (c) Byzantine faultload

to be possible. For instance, a collusion of clients can send requests only to some of the servers, preventing the requests from being processed, just like the attack done by Byzantine processes we study in Section 4.3. Also, malicious servers might send messages with future sequence numbers, which can not be discarded without risk of breaking safety properties. The service provided by BFT requires clients to wait for the reply to a request before issuing another one, thus constraining the possibility of a client to cause buffer overflows (but not a collusion of clients). BFT acknowledges the existence of the buffer overflow problem at *service level*, stating that the system has to bound the amount of *reply buffer* space by discarding the oldest replies (but storing enough information about the replies to inform clients that request them too late).

6 Conclusion

This paper presents the first study of the buffer overflow problem, and the correlated issue of (buffer) garbage collection, in intrusion-tolerant systems and protocols. The main purpose of the paper is to bring attention to this problem, and to the importance of handling it explicitly in works on intrusion tolerance. The motivation is the fact that papers in the area barely mention the problem, with the exception of BFT [5]. The argument in the paper suggests that these works are vulnerable to attacks targeting this problem, which might break either safety or liveness properties. In the latter case, systems would be vulnerable to denial of service attacks, a plague in current distributed systems.

References

[1] M. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, 2000.

[2] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. *SIGOPS Operating Systems Review*, 9(5):67–74, 1975.

[3] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 105–122, 1996.

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

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

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

[7] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183, Oct. 2004.

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

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

[10] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2006 CSI/FBI computer crime and security survey. Computer Security Institute, 2006.

[11] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[12] S. Kent and R. Atkinson. Security architecture for the Internet protocol. IETF Request for Comments: RFC 2093, Nov. 1998.

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

[14] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. IETF Request for Comments: RFC 2104, Feb. 1997.

[15] J. H. Lala, editor. *Foundations of Intrusion Tolerant Systems*. IEEE Computer Society Press, 2003.

[16] B. Lampson. Reliable messages and connection establishment. In S. Mullender, editor, *Distributed Systems*, chapter 10, pages 251–282. ACM Press / Addison-Wesley, 1993.

[17] B. Littlewood and L. Strigini. Redundancy and diversity in security. In P. Samarati, P. Rian, D. Gollmann, and R. Molva, editors, *Proceedings of the 9th European Symposium on Research Computer Security*, LNCS 3193, pages 423–438. Springer, 2004.

[18] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.

[19] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–213, 1998.

[20] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 568–577, June 2006.

[21] Y. Moses and G. Roth. On reliable message diffusion. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–128, Aug. 1989.

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

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

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

[25] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Procedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270. USENIX, Dec. 2002.