On the Effects of Finite Memory on 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, Universidade Federal de Santa Catarina
giuliana@lasige.di.fc.ul.pt mpc@di.fc.ul.pt lau@ppgia.pucpr.br pjv@di.fc.ul.pt

1. Introduction

Intrusion tolerance has been proposed as a new paradigm for computer systems security [2, 7]. 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.

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 due to temporary network partitions. In this scenario, attackers can force the filling of buffers, an effect often called a *buffer overflow* – not to be confused with C/C++ 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 showing that the problem appears at three levels: channels, protocol instances and service (Section 2). After presenting the problem, the paper studies buffer overflows with an I/T group communication primitive, inspired by the Rampart toolkit [5], and proposes the use of *repair nodes* to mitigate the problem (Section 3). An experimental evaluation with and without repair nodes is presented, allowing to assess in practice the effects of finite memory in a real, albeit simple, I/T system.

2. The Problem

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),

*This work was supported by the EC, through project IST-4-027513-STP (CRUTIAL) and Alban scholarship E05D057126BR, by the FCT through LASIGE and project POSI/EIA/60334/2004 (RITAS), and by CAPES/GRICES through project TISD.

[1]These latter buffer overflow 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.

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 potentially 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.

The problem appears at three levels: channels, protocol instances and service level.

Channels. 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., [4, 5]). The problem of reliable channels with finite memory is essentially the problem just described, which can be stated more formally as: *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.*

Protocol Instances. Many I/T systems run several instances of the same I/T protocol(s) concurrently. For instance, in [4, 5] 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: *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.*

Service. Most I/T systems replicate in a set of servers a *service*, which is accessed by a set of clients [1, 2, 3]. Clients send requests to the servers, which process the requests and send back replies. The problem is that a malicious client may send requests without acknowledging the reception of the replies, leaving those messages in the servers' buffers until they overflow: *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.*

3. 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 [5] and RITAS [4]. The objective is to propose a technique *–repair nodes–* to mitigate this problem when there are long communication delays or temporary disconnections. The system is formed by n processes, at most f of which are assumed to fail.

In order to study the buffer overflow problem in I/T protocols, we use a *message dissemination protocol* based on the *echo broadcast* proposed by Toueg [6]. That protocol guarantees two properties: (1) if a correct process sends a *data message*, all correct processes deliver that message; and (2) no two correct processes deliver two different data messages with the same identifier. The protocol, presented in Figure 1, satisfies the two properties if the channels have unlimited capacity (implying infinite memory). In the original echo protocol, when the first message is received, all correct processes reply with a message containing the same data; in our protocol each process replies with a message containing only the hash of this data, reducing the use of buffer space. 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*. In the second *when*, any subsequent $\langle$MULTICAST, INITIAL, p_i, $seq(m)$, *, *$\rangle$ message received by p_j from the same process p_i is ignored.

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

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

when p_j receives messages $\langle$MULTICAST, echo ,*, $seq(m)$, $H(m)\rangle$ from $(n-f)$ distinct processes **do**
  Accept m

Figure 1. Message dissemination protocol

3.1. Buffer Management

Suppose we have a system in which processes communicate using the echo multicast protocol in the previous section. 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 2 we showed the problems related to finite memory. 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. 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. 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, the total memory available in the system is limited, so we should avoid occupying it with several copies of the same data. Next we present a solution to optimize buffer management.

Message Storage We now present our strategy to store messages received by processes using the dissemination protocol described above. When a process receives the first message of a protocol instance, it creates a *context object* 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). Whenever a context is discarded, an object with the hash of the message, the sender identifier and the message identifier is stored in another buffer, called the *hash buffer*.

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* N_{rn} and using this value as an index of a vector V with subsets of the processes.

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 $\frac{3f+1}{f+1}$; if $N_{rn} = f+1$ and $n \gg f$ the gain tends to n/f. The responsibility of message buffering is shared by all processes but each one stores only a subset of the messages.

Message Recovery The purpose of using repair nodes is to allow processes that did not receive a message to *recover* it. The message recovery protocol has two parts: one to recover *echo* messages and another to recover *initial* messages. The recovery mechanism periodically analyzes each context in the *context buffer*. If a 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). 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 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 (Figure 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).

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*. The *context 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. 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 an *age-based collector* that 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.

3.2. 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 two different *buffer management policies*:

P1 – with repair nodes (our proposal): in this approach the responsibility of message buffering is shared by all processes. Messages are discarded from the buffers by the context and age-based collectors (see above).

P2 – without repair nodes: each message is stored by all processes and discarded only when all processes confirm its reception (since other processes may request its retransmission). If the buffer is full, the oldest messages are discarded by the age-based collector.

The two policies use the recovery protocol presented above. For policy P1 the number of repair nodes was $N_{rn} = f+1$. For both policies, $N_{recov} = 2$ and $T_{recov} = 3$ seconds. The experiments were executed with values of f from 1 to 5 and the number of processes was set to $n = 3f+1$, thus varied from 4 to 16. There were two different faultloads. 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.

The experiments were run on the Emulab environment [8], 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. 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.

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 evaluate the benefit of having this scheme (policy P1), instead of storing the messages in all processes that receive them (policy P2).

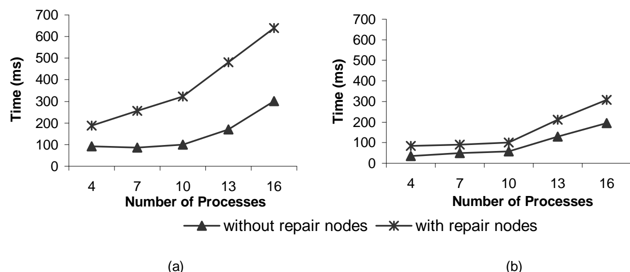

Figure 2. Time to discard the first data message (a) fail-stop and (b) Byzantine faultloads, with and without repair nodes

This 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 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 P1 and P2 is approximately proportional to N_{rn}/n, as also expected. The time to discard the first message with repair nodes (P1) 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. In all experiments we had better results when using the repair nodes scheme, i.e., with policy P1, 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.

References

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

[2] 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.

[3] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, Oct. 1998.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.