Wormhole-Aware Byzantine Protocols [*]

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

Abstract

This paper describes a new way of designing intrusion-tolerant protocols that rely, at certain points of their execution, on services provided by wormholes. Wormholes are enhanced components with stronger properties usually not available in the rest of the system. Our experience with this type of setting has shown that it is possible to design protocols with some interesting characteristics such as good performance, improved resilience, and independence from the FLP impossibility result.

1 Introduction

As society becomes more and more dependent on computer systems, attacks and intrusions perpetrated by malicious adversaries are important problems that need to be addressed in any IT infrastructure. In fact, current statistics published by CERT show that these problems are not disappearing, as indicated by the exponential grow on the number of reported incidents during the last decade [1].

Usually, protocols developed for these environments, in particular when there is unpredictable timeliness in the nodes and network, suffer in efficiency and/or determinism and/or liveness in order to tolerate malicious intrusions. In the paper we describe a new way of designing intrusion-tolerant protocols, also called Byzantine resilient protocols, which addresses this conflict. The fundamental characteristic of these protocols is that they rely on a wormhole at certain steps of their execution. Wormholes are enhanced components with properties usually not available in the rest of the environment [6]. For instance, in terms of security, the majority of the system might suffer from arbitrary (or Byzantine) failures while the wormhole can at most crash (i.e., intrusions are prevented from occurring in this component). As a consequence, the services provided by the wormhole can not be subverted, and they always return correct results (or nothing in case of a crash). This type of behavior, however, can not be naively assumed, but it has to be enforced during the construction of the system, using design principles such as architectural hybridization.

[*]This work was partially supported by the FCT through project POSI/CHS/39815/2001 (COPE), and the Large-Scale Informatic Systems Laboratory (LASIGE).

In the rest of our discussion we will consider an asynchronous Byzantine environment augmented with a specific kind of wormhole, named *Trusted Timely Computing Base* (TTCB) [5] (see Figure 1). A TTCB can be characterized as a secure real-time and fail-silent (crash) distributed component. In this setting, processes observe a typical computing environment – that we call the *payload system* – where they can experience undefined delays and arbitrary failures. However, in our approach, processes can call the TTCB at certain points of their execution, to perform correctly “small” crucial operations of the protocols. “Small” is an important objective since the wormhole should be kept as simple as possible, and with limited resources that have to be shared among all potential users.

To exemplify our approach, we will present a wormhole-aware consensus protocol. This protocol is relatively simple and has the following nice features. Even though it is completely time-free, the protocol is not bound by the FLP impossibility result because all the necessary time assumptions are concealed in the wormhole. This solution basically has the same merits as the Chandra and Toueg failure suspector abstraction [2], but without needing to detect failures, which is a relatively difficult task to accomplish in Byzantine environments. The protocol potentially performs well because it has a small number of rounds. Moreover, during runtime, it does not have to use public-key cryptography, a well-known and significant overhead in this type of protocols.

2 System Model

The architecture of the system is divided in two parts, the payload subsystem where processes are executed and the wormhole (see Figure 1). The payload subsystem en-

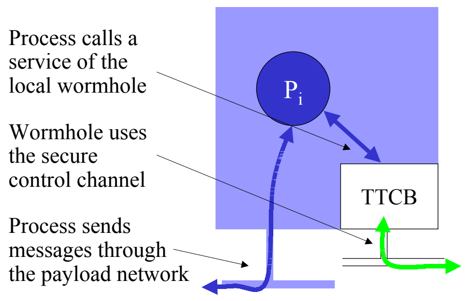
Process calls a service of the local wormhole
Wormhole uses the secure control channel
Process sends messages through the payload network
P_i
TTCB

Figure 1. Architecture of one node (the payload system is represented in dark and the wormhole in white).

compasses the usually available software of any computer, such as operating system and middleware libraries, and a payload network where processes can exchange messages. The wormhole is a distributed component with a private network called the *control channel*. It offers a limited set of services that can be used by the processes when needed.

Any entity running in the payload subsystem can experience unbounded delays and arbitrary failures. Therefore, a message will take an unknown interval to be delivered. Moreover, while in transit, it might be attacked by an adversary. We make the usual assumption that the adversary controls less than one third of the processes (i.e., $n \geq 3f + 1$).

The wormhole is built to be timely and secure. Consequently, once a service starts to be executed, it will take a well-know period for the answer to be available at the TTCB's interface (but it will take an arbitrary time to be delivered to the process). Even if an adversary manages to control the payload part of a node, she (or he) will not be able to subvert the local TTCB.

Payload channel assumptions We make the assumption that payload channels are *fair*, which means that if a process sends infinitely many messages to a single destination then infinitely many of those messages are correctly received. Therefore, if each pair of processes shares a symmetric key, it is possible to construct *secure channels* on the payload network with the following properties: a) if p and q are correct and p sends a message M to q, then q eventually receives M; b) if p and q are correct and q receives a message M with *sender(M)* = p, then M was sent by p and M was not modified in the channel [1].

Wormhole interface The only service used by the protocol is a low-level agreement on fixed sized values (20 bytes) called TBA. Processes propose a value and then they get a result. The service interface is:

error, value, prop-ok $\leftarrow$
$\leftarrow$ TTCB_TBA(eid, elist, aid, quorum, val)

The first three arguments are mainly used for identification: *eid* is the process identifier, *elist* is a list with the processes that will potentially call the service, and *aid* the agreement identifier. *quorum* defines the minimum number of proposed values *val* that must be received before the agreement can start. The result provided by the service is the *val* that was proposed by most entities (*value*). Other returned information is an *error*, and a mask *prop-ok* indicating the processes that proposed the decided value.

3 Solving Consensus

In the consensus problem a group of processes attempts to find a common value that is calculated using the originaly proposed values. We will solve a consensus with the following properties: *Validity* - if all correct processes propose the same value *v*, then any correct process that decides, decides *v*; *Agreement* - no two correct processes decide differently; *Termination* - every correct process eventually decides.

Before we start a more detailed description of the protocol, we would like to emphasize that the consensus problem does not become much easier simply because our solution utilizes a low level agreement service (the TBA). First, the difficulties created by the asynchronous setting continue to exist because any of, process execution, communication through the payload channel, and wormhole service invocations, might be delayed by an unknown amount of time. Second, both the node and the communication channels might experience Byzantine failures, which means that wrong, contradictory or malicious data can be received by the processes.

3.1 The Protocol

Each process running the protocol executes the algorithm 1. The arguments of the function are a list with the identifiers of the processes ($elist$), a unique consensus identifier (cid), and the value that will be proposed ($value_i$). The value can have an arbitrary number of bytes, ranging for instance from 1 bit (a "yes" or "no" agreement) to 10 MBytes (the contents of some file).

The protocol is organized in two phases. In the first phase, processes use the payload network to transmit the values to the other processes (Lines 4-8). Since communication is done with secure channels, an adversary that controls the network is unable to change the message contents, and the right values are received. Processes stay in

[1]One can build these channels by adding Message Authentication Codes (MAC) to the messages and using re-transmissions to ensure delivery.

Algorithm 1 Consensus (executed by every p_i)

```
1  function consensus(elist, cid, $value_i$)
2  hash-v ← ⊥                                   {hash of the decided value}
3  bag ← ⊥                                      {bag of received values}

4  broadcast(B-value, i, $value_i$)              {payload channel}
5  repeat
6      receive(B-value, k, $value_k$)
7      bag ← bag ∪ {$value_k$}
8  until (bag has $2f+1$ values from different processes)
9  activate task(T1, T2)                         {start two concurrent tasks}

10 task T1:
11 v ← most_Common_Value(bag)
12 out ← TTCB_TBA(eid, elist, cid, $2f+1$, Hash(v))
13 if (at least $f+1$ proposed the same value) then
14     hash-v = out.value
15 else
16     return (default-value)

17 task T2:
18 when (receive(Decide, k, $value_k$)) do
19     bag ← bag ∪ {$value_k$}
20 when (hash-v ≠ ⊥) and ($\exists_{value_k \in bag}$ : Hash($value_k$) = hash-v))
   do
21     broadcast (Decide, i, $value_k$)
22     return ($value_k$)
```

this phase until $2f + 1$ values have been collected from different senders, which ensures that there is a majority of values belonging to correct processes (at least $f + 1$) in the various bags.

In the second phase, processes agree on one of the values, using the wormhole to accomplish this task. Processes start by selecting the most common of the stored values in the bag (Line 11). This simple condition guarantees that all correct processes choose the same value if they all had identical proposals. On the other hand, if correct processes had different proposals, then they can pick distinct values and even values submitted by a malicious process (it all depends on the network delays and on the existence of collusion among the processes controlled by the adversary).

Next, processes call the TBA service of the wormhole with an hash of the chosen value, $Hash(v)$, and wait for a decision (Line 12). A majority of such calls comes from correct processes because the $quorum$ parameter is set to $2f + 1$. When the TBA returns, all processes obtain the same results, and then they perform a few tests to determine how they should terminate. Basically, there are two possible outcomes, depending on the original proposals (*prop-ok* is used to make this test at Line 13). If all correct processes had the same original value, then they can decide immediately since this value will be supported by at least $f+1$ votes (Lines 14 and 20-22). Otherwise, either a default value is chosen (Line 16) or another value that by luck had $f + 1$ or more votes (Lines 14 and 18-22).

It is relatively easy to demonstrate that the protocol verifies all properties of consensus.

4 Conclusions and Future Directions

The paper introduces a new way of designing intrusion-tolerant protocols. The distinctive characteristic of our approach is that processes continue to execute in a Byzantine asynchronous environment, but they can call the wormhole' services at certain steps of their execution, to perform correctly "small" crucial operations of the protocols.

In the paper we have described an example protocol that solves consensus problem. This protocol has some interesting characteristics: good performance since it has a small number of rounds and does not need to use public-key signatures; and it is not constrained by the FLP impossibility result since all synchrony assumptions can be hidden inside the wormhole. In other problems we have shown that is possible to construct wormhole-aware protocols that have better resilience than their counterparts, for example reliable multicast [3] and state-machine replication [4].

In the future, there are several avenues that can be explored:

- Definition of weaker wormholes and services that allow the design of protocols with good properties.
- More design, implementation and evaluation of other wormholes and protocols.
- Better formalization and analysis of the wormhole model.

References

[1] CERT Coordination Center. CERT/CC Statistics 1988-2003. http://www.cert.org/stats/cert_stats.html, March 2004.

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

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

[4] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in pratical distributed systems. Manuscript, March 2004.

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

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