**Distributed Computing manuscript No.**
(will be inserted by the editor)

# Low Complexity Byzantine-Resilient Consensus ⋆

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

[1] Faculdade de Ciências da Universidade de Lisboa, 1749-016 Lisboa, Portugal (e-mail: {mpc,nuno,pjv}@di.fc.ul.pt)
[2] Pontifícia Universidade Católica do Paraná, 80215-901 Prado Velho, Brasil (e-mail: lau@ppgia.pucpr.br)

**Summary.** The application of the tolerance paradigm to security – *intrusion tolerance* – has been raising a reasonable amount of attention in the dependability and security communities. In this paper we present a novel approach to intrusion tolerance. The idea is to use privileged components – generically designated by *wormholes* – to support the execution of intrusion-tolerant protocols, often called Byzantine-resilient in the literature.

The paper introduces the design of wormhole-aware intrusion-tolerant protocols using a classical distributed systems problem: consensus. The system where the consensus protocol runs is mostly asynchronous and can fail in an arbitrary way, except for the wormhole, which is secure and synchronous. Using the wormhole to execute a few critical steps, the protocol manages to have a low time complexity: in the best case, it runs in two rounds, even if some processes are malicious. The protocol also shows how often theoretical partial synchrony assumptions can be substantiated in practical distributed systems. The paper shows the significance of the TTCB as an engineering paradigm, since the protocol manages to be simple when compared with other protocols in the literature.

**Key words:** Byzantine fault tolerance – intrusion tolerance – distributed systems models – distributed algorithms – consensus

## 1 Introduction

Attacks and intrusions perpetrated by hackers are important security problems faced by any computer infrastructure. These malicious actions fall into the category of arbitrary faults, which sometimes have been called "Byzantine" faults [23]. The work reported in the paper comes from a recently finished project that investigated the application of the fault tolerance paradigm to enhance the security of systems [1,36]. In the project we explored two recent key ideas on distributed systems architecture. The first is *wormholes*, enhanced components which provide processes with a means to obtain a few simple privileged functions and/or channels to other processes, with "good" properties otherwise not guaranteed by the "normal" environment [34]. For example, a wormhole might provide timely or secure functions and communication in, respectively, asynchronous or Byzantine systems. The second key idea is *architectural hybridization*, a well-founded way to substantiate the provisioning of those "good" properties on "weak" environments. For example, if we assume that our system is essentially asynchronous and Byzantine, we should not simply (and naively) postulate that parts of it behave in a timely or secure fashion. Instead, those parts should be built in a way that our claim is guaranteed with high confidence.

Consensus is a classical distributed systems problem with both theoretical and practical interest. Over the years, other problems have been shown to be reducible or equivalent to consensus, for instance, total order broadcast [20]. Consensus has been applied to various kinds of environments, with distinct time assumptions and different types of failures, ranging from crash to arbitrary (see [17] for a survey of early work). On asynchronous environments, it was shown to be constrained by the FLP result, which says that it is impossible to solve consensus deterministically if failures can occur [18]. Several researchers have proposed ways to circumvent this limitation, e.g., by using randomization techniques [30,4,5], by using failure detectors [8,25,21], and by using partial-synchrony, i.e., by making weak synchrony assumptions [16, 13]. However, all these approached have practical shortcomings when the objective is to tolerate Byzantine faults. Randomized protocols usually need a large number of message rounds, failure detectors can detect only a subset of all possible failures and synchrony assumptions are hard to substantiate.

The present paper shows how a wormhole can be utilized to support the execution of a Byzantine-resilient (or intrusion-tolerant) protocol. More specifically, the paper presents a consensus protocol based on a specific kind of wormhole, a de-

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

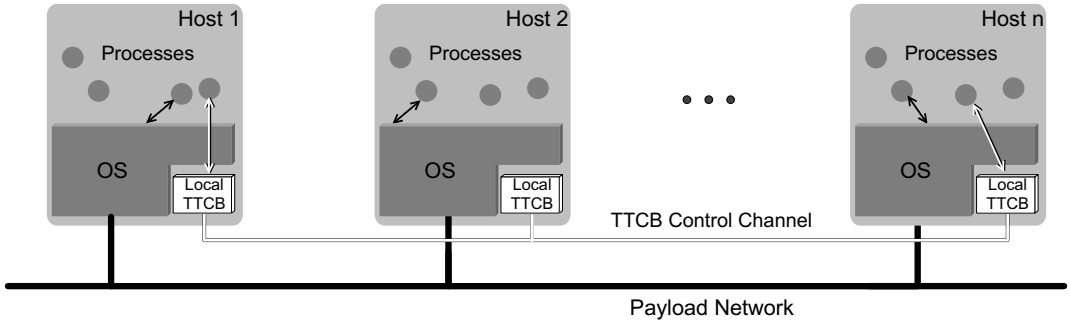


**Fig. 1.** Architecture of a system with a TTCB.

vice called *Trusted Timely Computing Base* (TTCB). Technically, the TTCB is a secure real-time and fail-silent (crash) distributed component. Applications implementing the consensus protocol run in a "normal" system, which puts no restrictions on the type of failures that might happen and has no time bounds on the execution of operations or communication, i.e., a typical asynchronous Byzantine system. Normally, applications for these environments would suffer in efficiency and/or determinism. However, the TTCB is locally accessible to any process, and the touchstone of our approach is that at certain points of their execution applications can rely on it to execute correctly (small) crucial steps of the protocols. A graphical representation of a networked architecture with a TTCB wormhole can be observed in Figure 1. The reader intrigued by the feasibility of building a wormhole might wish to refer to [12, 11] for a description of the implementation of a TTCB using architectural hybridization[1]. The consensus protocol relies to most extent on a TTCB service called *Trusted Block Agreement Service* (TBA), which essentially makes an agreement on the values proposed by a set of processes. In this paper we also assume that the processors are partially synchronous, i.e., that the processing delays stabilize after an unknown interval of time.

The main contributions of the paper are the following. Firstly, it presents the design of a consensus protocol based on the TTCB wormhole and, more specifically, on a service that securely makes agreement on fixed and limited size values (TBA). Secondly, the paper shows how, often theoretical, partial synchrony assumptions can be well substantiated in practical distributed systems. Thirdly, the protocol presented has a low time complexity: in the best case, it runs in two rounds, even if some processes are malicious, mostly due to the strong properties enforceable in a component like the TTCB. Finally, the protocol shows the significance of the TTCB as an engineering paradigm, since the protocol manages to be simple when compared, e.g., with other protocols based on weak synchrony assumptions, like those in [16].

The remainder of the paper is organized as follows: The system model and the TTCB are presented in Section 2. The TTCB services used by the protocol are introduced in Section 3. The consensus problem and the protocol are described in Section 4. Section 5 evaluates the protocol in terms of the time and message complexities. Section 6 discusses related work and Section 7 concludes the paper.

[1] A software implementation of the TTCB is available online at http://www.navigators.di.fc.ul.pt/software/ttcb/.

## 2 System Model

### *2.1 System Architecture*

The architecture of the system can be seen as a classical Byzantine asynchronous distributed system (designated here by *payload system*) augmented with the TTCB wormhole. In Figure 1, the parts in white constitute the augmented subsystem, which in security terminology would be called a real-time distributed security kernel. All the applications and protocols are executed in the payload system, except for some calls to the wormhole.

In each host there is a *local TTCB*, which is a small component conceptually separated and protected from the remainder of the host (the operating system and other software). The local TTCBs are all interconnected by a *control channel* which is assumed to be secure. Collectively, the control channel and the local TTCBs are called *the TTCB* [12]. The payload system is composed by the usual software available in hosts (such as the operating system and applications) and the *payload network* (the usual network allowing communication among the various nodes, e.g., Ethernet/Internet). Throughout the paper we assume that the protocol is executed by processes in the hosts, which communicate through the payload network and call the TTCB to execute one of its services.

### *2.2 Fault and Time Models*

Fault-tolerant systems are usually built using either arbitrary or controlled failure assumptions. Arbitrary failure assumptions consider that components can fail in any way, although in practice constraints have to be made (e.g., that less than one third of the processes fail). These assumptions are specially adequate for systems with malicious faults –attacks and intrusions [1]– since these faults are induced by intelligent entities, whose behavior is hard to restrict or model. Controlled failure assumptions are used for instance in systems where components can only fail by crashing. *Architectural-hybrid failure assumptions* bring together these two worlds: some components are constructed to fail in a controlled way, while others may fail arbitrarily. In this paper we assume such a hybrid fault model, where all system components are assumed to fail arbitrarily, except for the TTCB that is assumed to fail only by crashing [2].

[2] There is some research on *hybrid fault models*, starting with [27], that assumes different failure type distributions for system

The payload system is *insecure* and in essence *asynchronous*. In relation to insecurity, this means that processes running in the payload system – including the processes that execute the protocol in this paper – can fail in an arbitrary way due to benign faults or to the action of an attacker. They can, for instance, give invalid information to the TTCB, stop communicating, or start colluding with other malicious processes. The payload system is mostly asynchronous since it has unbounded and unknown message delivery delays, and unbounded and unknown local clock drift rates. However, we make a weak synchrony assumption about the delays in the processors: for each execution of the protocol there is an unknown *processors stabilization time* (PST) such that the processing delays are bounded from time PST onward. This synchrony assumption is weak in the sense that it is about the processors instead of about the network, which is important since the delays inside the processors are much more deterministic than in a network. This property has to be distinguished from the typical synchrony assumptions in the literature, like those concerning unreliable failure detectors, which can only be implemented in a real system with some synchrony assumptions about the network.

The TTCB has two fundamental characteristics: it is *secure* and *synchronous*. In relation to security, it is built to fail only by crashing, albeit inserted in a system where arbitrary, even malicious faults do occur. The component is expected to execute its services reliably, even if malicious hackers manage to attack the hosts with local TTCBs and the payload network. The TTCB is also a synchronous subsystem capable of timely behavior, in the line of the precursor Timely Computing Base work [35]. In other words, it is possible to determine a (maximum) delay for the execution of the TTCB services. The local TTCBs clocks are synchronized (see Section 3).

### *2.3 TTCB Implementation*

The TTCB has an abstract specification that can be implemented in different ways. The current design, which is based on COTS components, was reported elsewhere [11, 12]. This section briefly discusses this implementation for completeness.

The system is composed by a set of standard PCs, each one with two independent LAN adapters for Fast-Ethernet (see Figure 2). The TTCB control channel is implemented in one of the LANs and the payload network on the other. PCs run RTAI which is an extension of Linux that supports the execution of real-time applications [9]. RTAI modifies the Linux kernel in such a way that a real-time executive takes control of the hardware to enforce the timely behavior of some tasks. These RT tasks are special Linux loadable kernel modules (LKMs), which means that they run inside the kernel (see figure). The scheduler was changed to handle these tasks in a preemptive manner and to be configurable to different

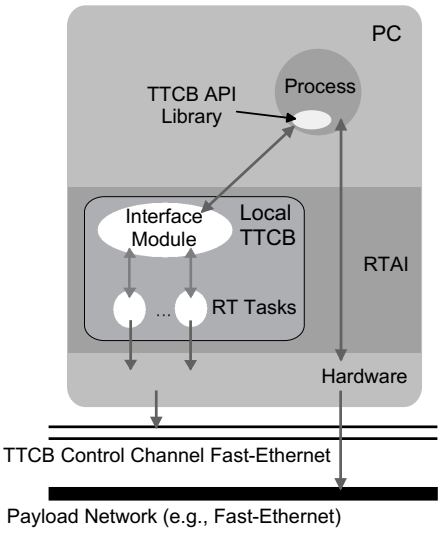


**Fig. 2.** COTS-based TTCB implementation.

scheduling disciplines. Linux runs as the lowest priority task and its interruption scheme was changed to be intercepted by RTAI.

The local TTCB is implemented by a LKM and by a number of RT tasks. This LKM – the TTCB Interface Module – handles the calls from the processes. It does not provide real-time guarantees since it is part of the TTCB interface, i.e., it is the border between the asynchronous and the synchronous parts of the system. All operations with timeliness constrains are executed by RT tasks. A local TTCB always has at least two RT tasks that handle its communication: one to send messages to the other local TTCBs and another to receive and process incoming messages. The API functions are defined in libraries and communicate with the local TTCB using RT FIFOs. Currently there are C and Java libraries available.

From the point of view of security, RTAI is very similar to Linux. Its main "vulnerability" is the ability of the superuser to control any resource in the system. This vulnerability is usually reasonably easy to exploit once the attacker is logged in the machine, e.g., by using race conditions. Recently, several Linux extensions appeared that try to compartmentalize the power of the superuser. Linux *capabilities* [33], which are already part of the kernel, are one of those mechanisms. These capabilities are privileges or access control lists associated with processes, allowing a fine grain control on how they use certain objects.

A secure TTCB can be constructed by performing the following steps. First, one needs to protect the operating system and the TTCB binary files executed during the system bootstrap. This task is accomplished by booting the system from a read-only device, such as a ROM or CD-ROM. After the kernel boots and the local TTCB starts to run, the kernel is sealed, i.e., the possibility of directly modifying the kernel memory or inserting code in it, is disabled even for processes with superuser privileges. This is achieved by removing some capabilities from the *capability bounding set*, so they can never be granted to any process until the next reboot (see details in [12]).

The TTCB control channel must also be protected. Firstly, we must guarantee that it is not possible to read or to write

nodes. These distributions would be hard to predict or constrain in the presence of malicious failures introduced by, for example, hackers. Our work is not related to that research but to the idea of *architectural hybridization*, in the line of works such as [29, 37], where failure assumptions are in fact enforced by the architecture and the construction of the system components, and thus substantiated.

the control channel access point in a host. This is forced by removing the access to the device driver so that only code inside the kernel (the local TTCB) can use it. Next, the network medium has to be secured from physical access. This protection is simply assumed, since we are considering a short-range, inside-premises closed network, connecting a set of servers inside a single institution, with no other connection. We assume that the attacker comes from the Internet through the payload network, without physical access to the servers or control network hardware.

The real-time behavior of the local TTCB is guaranteed by RTAI. By protecting the kernel we also prevent attacks against this real-time behavior, since the local TTCB runs inside the kernel. If the Fast-Ethernet is secured as described above, the bounded delay assumption of the control channel can be ensure if: (1) only one host is connected to each port to avoid collisions; and (2) the traffic load is controlled (see [7] for details). The first requirement is trivial. The second is guaranteed by an access control mechanism, that accepts or rejects the execution of one service taking in account the availability of resources, i.e., buffers and bandwidth.

In the future, we envisage other TTCB designs, for instance based on tamperproof hardware (e.g., a PC104 board with its own processor and memory) and wide-area networks such as a virtual private networks (VPN) based on ISDN, Frame Relay or ATM connections. If we assume that the public telecommunications network is not eavesdropped then a VPN already provides a private channel. Additional security can be obtained using secure channels, e.g., encrypting the TTCB communication.

### *2.4 Communication Model*

The protocol relies on channels that abstract some of the communication complexity. Each pair of processes $p$ and $q$ is interconnected in the payload network by a *secure channel*, defined in terms of two 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 sent by $p$ and M was not modified in the channel. [3]

Each pair of correct processes is assumed to share a symmetric key known only by the two. With this assumption, the two properties above can be easily and efficiently implemented. Eventual reliability is obtained by retransmitting the messages periodically until an acknowledgment is received. Message integrity is achieved by detecting the forgery and modification of messages through the use of Message Authentication Codes (MACs) [26]. A MAC is basically a cryptographic checksum obtained with a hash function and a symmetric key. They are usually considered to be three orders of magnitude faster to calculate than digital signatures. A process adds a MAC to each message that it sends, to allow the receiver to detect forgeries and modifications. Whenever such detection is made, the receiver simply discards the message, which will be eventually retransmitted if the sender is correct.

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

## 3 TTCB Services

The TTCB provides a limited number of security- and time-related services [12]. Here we introduce only the three services used in the protocol presented in the paper.

The *Local Authentication Service* allows processes to authenticate the local TTCB, obtain an unique identifier (called *eid*) and establish a shared symmetric key with it. The objective of this key is to protect the communication between the process and the wormhole. For example, when a result arrives from a TTCB service, the process can use the key to verify the authenticity and integrity of the data. If the key is discovered by an attacker, a personification attack becomes possible, and consequently the process has to be considered failed. The process has also to be considered failed if there is a denial-of-service (DoS) attack in its host[4]. In terms of assumptions, the local authentication service substantiates the assumption that the communication among processes and the TTCB is reliable. We consider that every process executing a protocol during its initialization called the local authentication service and obtained an *eid*. The execution of the local authentication service is the only moment when the protocol uses public-key cryptography [12].

The *Trusted Absolute Timestamping Service* provides globally meaningful timestamps, since the local TTCB clocks are synchronized (the synchronization protocol runs inside the TTCB). In practice this service provides a clock which is available at all hosts with a local TTCB. This clock is also secure, i.e., an attacker can not modify it.

### *3.1 Trusted Block Agreement Service*

The main service used by the consensus protocol is the *Trusted Block Agreement Service*, or simply *TBA Service*. This service delivers the result obtained from an agreement on the values proposed by a set of processes. All payload processes receive the same result from the TTCB, since the TTCB is secure. The values are blocks with small size, 20 bytes in the current implementation. Additionally, the TTCB resources are limited so this service should be used only to execute critical steps of protocols, which run mostly outside the wormhole.

The TBA service is formally defined in terms of the three functions *TTCB_propose*, *TTCB_decide* and *decision*. A process *proposes a value* when it successfully calls *TTCB_propose*. If for some reason the proposal is not accepted, an error is returned by the TTCB (in this case the value was not proposed, e.g., for property TBA4). A process *decides a result* when it calls *TTCB_decide* and receives back a result (*TTCB_decide* is non-blocking and returns an error if that execution of the service did not terminate). The function *decision* calculates the result in terms of the inputs of the service.

[4] If an attacker manages to log in a host, it can attempt a DoS attack by calling the TTCB at a fast rate. Some of its requests are accepted but others are discarded by the access control mechanism mentioned in Section 2.3. Neither the deterministic behavior of the TTCB control channel nor the other local TTCBs are affected. However, a process in that host might not be able to access the TTCB, therefore it has to be considered failed. This case is equivalent to a malicious process consuming all CPU time in the host.

The *result* is composed of a value and some additional information that will be described below. Formally, the TBA service is defined by the following properties:

- *TBA1 Termination.* Every correct process eventually decides a result.
- *TBA2 Integrity.* Every correct process decides at most one result.
- *TBA3 Agreement.* No two correct processes decide differently.
- *TBA4 Validity.* If a correct process 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 implementation of the TBA inside the TTCB is briefly introduced later in Section 5.1. Here we present its API, which consists of two function calls:

tag,error ←TTCB_propose(eid, elist, tstart, decision, value)

value,proposed-ok,proposed-any,error ←TTCB_decide(tag)

The parameters have the following meanings. *eid* is the identification of a process before the TTCB, provided by the Local Authentication Service. *elist* is an array with the eid's of the processes involved in the TBA [5]. *tstart* is a timestamp that indicates the instant when proposals for the TBA are no longer accepted and the TBA can start to run inside the TTCB. When the local TTCB receives a proposal, it reads the clock and compares it with the value of *tstart*. If the clock value is later than *tstart*, it returns an error, otherwise the proposal is processed in the normal way. The objective of this test is to prevent malicious processes from postponing TBAs indefinitely. *decision* indicates the decision function used to calculate the result. There is a set of decision functions but the protocols in this paper use only one that returns the value proposed by more processes, designated *TBA_MAJORITY*. If there are several values with the same number of proposals, one is chosen. *value* is the value being proposed. The TTCB knows that proposals pertain to the same TBA when *(elist, tstart, decision)* are the same.

*TTCB_ propose* returns a *tag*, which is used later to identify the TBA when the process calls *TTCB_ decide*, and an *error* code. Notice that, even if a process is late and calls *TTCB_propose* after *tstart*, it gets the *tag* and later can get the result of the agreement by calling *TTCB_ decide*. This second function returns four things: (1) the value decided; (2) a mask *proposed-ok* with bits set for the processes that proposed the value that was decided; (3) a mask *proposed-any* with bits set for the processes that proposed any value (before *tstart*); and (4) an error code. Notice that the *result* of the service mentioned in the TBA definition (properties TBA1-TBA5) is composed by *(value, proposed-ok, proposed-any)*.

[5] Notice that we may use "TBA" to denote "an execution of the TBA service", not the service itself.

## 4 Consensus

This section describes a consensus protocol tolerant to Byzantine faults. For presentation simplicity, we start by explaining how to reach consensus on a value with a small number of bytes, and then this result is extended by removing this limitation.

The consensus protocol utilizes as building block the TBA service. The reader however, should notice that, as tempting as it might be, it is *not* possible to solve the consensus problem in the payload system simply by using the TBA service of the TTCB. In fact, the problem does not become much simpler because the protocol still needs to address most of the difficulties created by a Byzantine asynchronous environment. For instance, since the protocol runs in the asynchronous part of the system, it can not assume any bounds on the execution of the processes, on the observed duration of the TTCB function calls, or on the message transmission times. Moreover, since processes can be malicious, this means that they might provide incorrect values to the TTCB or other processes, or they may delay or skip some steps of the protocol. What we aim to demonstrate is that the 'wormholes' model, materialized here by the TTCB, allows simpler solutions to this hard problem.

### *4.1 Consensus Problem*

The consensus protocol is executed by a finite set of $n$ processes $P = \{p_1, p_2, ...p_n\}$. The protocol tolerates up to $f = \lfloor \frac{n-1}{3} \rfloor$ faults. This has been proved to be the maximum number of faulty processes for consensus in asynchronous systems with Byzantine faults [5].

The problem of consensus can be stated informally as: how do a set of distributed processes achieve agreement on a value despite a number of process failures? There are several different formal definitions of consensus in the literature. In the context of a Byzantine fault model in asynchronous systems, a common definition [16, 25, 21] is:

- *CS1 Validity.* If all correct processes propose the same value *v*, then any correct process that decides, decides *v*.
- *CS2 Agreement.* No two correct processes decide differently.
- *CS3 Termination.* Every correct process eventually decides.

The Validity and Agreement properties must always be true otherwise something bad might happen. Termination is a property that asserts that something good will eventually happen. In the case all correct processes propose the same value, Validity guarantees that it is the value chosen, even in the presence of alternative malicious proposals. If correct processes propose different values, the consensus protocol is allowed to decide on any value, including on a value submitted by a malicious process. In systems with only *crash faults*, the Validity property can be stated in a more generic form: "if a correct process decides *v*, then *v* was previously proposed by some process". However, this definition is not adequate with Byzantine/arbitrary faults because a failed process does not just crash, as a matter of fact, usually it can be impersonated.

Another common definition of consensus for Byzantine settings is *vector consensus* or *interactive consistency*, in which processes agree on a vector of values proposed by a subset of the processes involved [15, 3].

### *4.2 Block Consensus Protocol*

The *block consensus* protocol reaches consensus on a value with a limited number of bytes. When compared with other Byzantine-resilient consensus protocols, block consensus is quite simple since most of its implementation relies on the TBA service of the TTCB, and no information has to be transmitted through the payload channel. Nevertheless, it serves to illustrate two interesting features of our system model. First, it demonstrates that it is possible to construct a consensus protocol capable of tolerating arbitrary attacks based on an agreement protocol that was developed under the crash fault model. Second, it shows how a protocol running under the asynchronous model can interact with one running synchronously (in the TTCB).

The protocol is presented in Algorithm 1. The arguments are the list of the $n$ processes involved in the consensus (*elist*), a timestamp (*tstart*), and the value to be proposed (*value*). *tstart* has to be the same in all processes. For the participants, this requirement is similar to what is observed in other consensus protocols where all processes have to know in advance a consensus identifier. However, the identifier conveys a meaningful absolute time to the TTCB: processes despite being time-free, can agree on a value obtained from the Trusted Absolute Timestamping service to synchronize their participation to the consensus. The number of bytes of *value* should be the same as the size imposed by the TBA service (currently 20 bytes). In case it is smaller, padding is done with a known quantity (e.g., with zero). The number of processes which can fail is $f = \lfloor \frac{n-1}{3} \rfloor$, as stated above.

The protocol works in rounds until a decision is made. In every round, each process proposes a value to the TBA (line 4) and gets the result (lines 5-7). In each round the value decided by TBA is the value proposed by most processes (decision *TBA_MAJORITY*). All correct processes get the same result of the TBA since the TTCB is secure[6]. The protocol terminates when one of the conditions is satisfied (line 10):

1. *at least* $f + 1$ *processes proposed the same value* $v$: this condition implies that at least one correct process proposed *v*. Therefore, either (1) all correct processes proposed *v* or (2) not all correct processes proposed the same value. In both cases, the protocol can terminate and decide *v*.
2. *at least* $2f + 1$ *processes proposed a value but no subset of processes with the same value has a size larger than* $f$: this condition implies that some correct processes proposed distinct values. In this case, the protocol can terminate and decide on any value. For example, our implementation will choose the most proposed value, if it exists.

Both conditions can be tested using the two masks returned by *TTCB_decide*. The first one is constructed with the *proposed-ok* mask and the second one can be evaluated with the *proposed-ok* and the *proposed-any* masks (Section 3.1). The TBA execution starts in one of two conditions: when all processes have proposed a value; or when time reaches *tstart*. Block consensus assumes that eventually there is a round when *enough* processes manage to propose to the TBA before *tstart*. 'Enough' here is defined in terms of the two conditions that allow the protocol to terminate. The algorithm keeps retrying until this happens (lines 3-10).

**Algorithm 1** Block consensus protocol.

```
1  function consensus(elist, tstart, value)
2  round ←0;                                              {round number}
3  repeat
4     out_prop ←TTCB_propose(eid, elist, tstart, TBA_MAJORITY, value);
5     repeat
6        out_dec ←TTCB_decide(out_prop.tag);
7     until (out_dec.error ≠ TBA_RUNNING);
8     tstart ←tstart + T * func(α, round);                {α ∈ [0, 1[}
9     round ←round+1;
10 until (f + 1 processes proposed the same value) or (2f + 1 processes proposed);
11 decide out_dec.value;
```

The *tstart* of the next round is calculated by adding a quantity to the previous *tstart*, computed using constants $T$ and $\alpha$, and function *func* (line 8): *func* is a monotonically increasing function of *round*, where $\alpha$ controls the slope, $\alpha \in [0, 1[$. For example, linear ($func \equiv 1 + \alpha * round$), or exponential ($func \equiv (1 + \alpha)^{round}$). Thus, by increasing the period of retry upon each repetition, we will eventually manage to get enough processes to propose. There is an interesting tradeoff here: with a larger *tstart* the probability of termination in real systems increases, since more time is given for proposals; on the other hand, if one process is malicious and does not propose, then a larger *tstart* will delay the execution of the TBA service, and consequently the consensus protocol. Incidentally, note that processes, being time-free, are totally unaware of the real-time nature of *tstart*, they just deterministically increase an agreed number, which is only meaningful to the TTCB.

At this moment, the reader might ask the following questions: Why run several agreements inside the TTCB in order to make a single consensus outside it? Running a single TBA is not enough? The answer 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 process calls *TTCB_propose* it provides a *tstart*, i.e., a timestamp that indicates the TTCB the instant when no more proposals

[6] The reader might wonder if a malicious process might break the protocol by calling *TTCB_propose* in line 4 with a subset of *elist*, *elist'*. Apparently this would give different results of the TBA for (1) the processes in *elist'* and (2) the processes in *elist* but not in *elist'*. In reality, this would not happen. The TTCB has the notion of a "TBA execution", which is uniquely identified by three parameters: *tstart*, *elist* and *decision* (see Section 3.1). Therefore, the effect of a malicious process providing a different *elist*, is the beginning of a second TBA execution, identified by *(tstart, elist', decision)*, completely independent of the first execution, identified by *(tstart, elist, decision)*. There can be no interference between the two TBA executions.

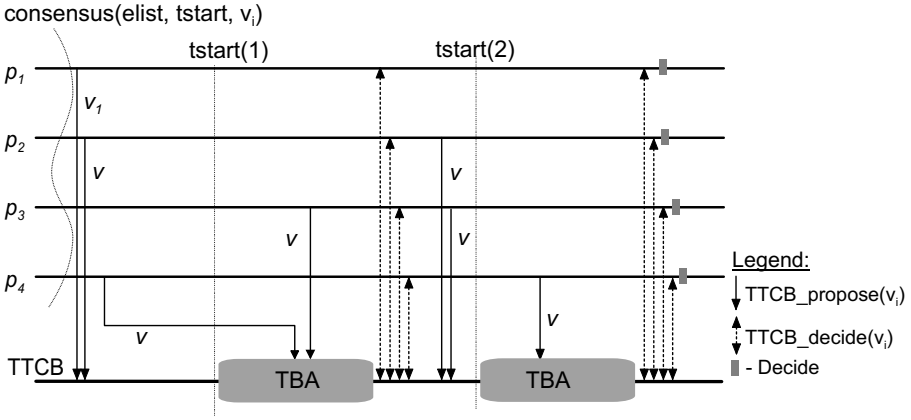


**Fig. 3.** Block Consensus protocol example execution (with *n=4* and *f=1*).

are accepted to the TBA identified by the arguments *(elist, tstart, decision)*. The process that calls *TTCB_propose* is in the asynchronous part of the system therefore we can never assume that the process will call *TTCB_propose* before instant *tstart*, regardless of the value of this parameter. The consequence of this to the consensus protocol is that each round any number of processes 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" processes manage to propose before *tstart*, i.e, until the condition in line 10 is satisfied.

Figure 3 illustrates an execution of the protocol in a system with four processes where $p_1$ is malicious. In the example, $p_1$ and $p_2$ are able to propose on time for the first TBA. $p_4$ starts on time, but is delayed for some reason (e.g., a scheduling delay) and proposes after *tstart(1)*. Therefore, it will get an error from the TBA service, and its value will not be considered in the agreement. $p_3$ is also delayed, and only starts to execute after *tstart(1)*, and consequently, its proposal is also disregarded. When the TBA finishes, all processes get the result, which in this case will be based on the proposals from $p_1$ and $p_2$. Since $p_1$ is malicious, it attempts to force an incorrect decision by proposing $v_1$ that is different from the value of the correct processes (which is $v$). Nevertheless, since none of the conditions is satisfied (line 10), another round is executed. Here, process $p_1$ skips the proposal step, but two correct processes manage to propose before *tstart(2)*. In the end, all correct processes will be able to decide, since the first condition will be true.

The correctness of the protocol is proved in Appendix A.

### *4.3 General Consensus Protocol*

For presentation simplicity, we first described the block consensus protocol, which achieves agreement on a data value with at most the size of the TBA service block. This section presents a consensus protocol without this limitation. The *general consensus* protocol makes use of the payload channel to multicast the values being proposed, and then utilizes the TBA service to choose which value should be decided. The number of processes which can fail is also $f = \lfloor \frac{n-1}{3} \rfloor$.

The protocol is presented in Algorithm 2. The arguments have the same meaning as in the block consensus. Each process starts by initializing some variables ($\emptyset$ is the empty bag, and $\perp$ is a value outside the range of valid hashes), and then it multicasts the value through secure channels to the other processes (line 6). Next, the protocol works in two phases, where it runs a minimum of one round in the first phase, but depending on the values and on the timing of the proposals, it may need several rounds in both phases.

In the *first phase* processes propose to the TBA a *hash* of their own value (line 11). A secure hash function is a one-way function assumed infeasible to invert, which compresses its input and produces a fixed sized digest (e.g., algorithm SHA gives a 20 byte output), that we will for simplicity call the *hash* [26] [7]. It is assumed that it is infeasible to find two texts that yield the same hash. This phase and the protocol both terminate if $f + 1$ processes propose the same hash to the TBA (line 19). In this case, the value decided is the one that corresponds to that hash (lines 20, 23, and 26). Since $f + 1$ proposed the hash, then at least one of the processes has to be correct. Consequently, it is safe to use that value as the decision (the argument is equivalent to the first condition of block consensus). Moreover, since a correct process always starts by multicasting its value through reliable channels, then we can be sure that eventually all correct processes will receive the value, and will be able to terminate.

The protocol enters the *second phase* when $2f + 1$ processes proposed a hash but no subset greater than $f$ proposed the same one (lines 17-18). This situation only happens when the correct processes do not have the same initial value. In this case the definition (Section 4.1) allows any value to be chosen. The simpler solution would be to choose a pre-established value, e.g., zero. However, it is more interesting to make the protocol agree on one of the various proposed values. This is the purpose of the second phase.

The second phase uses a rotating coordinator scheme [31]. In each round a different process is the coordinator ($coord = round \bmod n$), and then its value is selected as the (potential) decision. Processes pick the value of the current coordinator to propose it to the TBA. If this value is not available (for instance, because it was delayed or the coordinator crashed), then it is necessary to choose another value. In our case, we decided to use a simple deterministic algorithm where a process goes through the *elist* until it finds the first process whose

[7] The size of the value of the TBA service is 20 bytes which is precisely the size of a currently considered 'secure' hash (e.g., SHA has 20 bytes).

**Algorithm 2** General consensus protocol.

```
1  function consensus(elist, tstart, value)
2  hash-v ←⊥;                                   {hash of the value decided}
3  bag ←∅;                                      {bag of received messages}
4  round ←0;                                    {round number}
5  phase ←1;                                    {protocol phase}
6  multicast(elist, tstart, value) to processes in elist;   {send value
   through payload channel}

7  loop
8    repeat                                     {phase1: use my value}
9      if (phase = 2) then  {phase 2: choose value from process}
10       value ←{M.value : coord = (round mod n) ∧
         M=nextSenderMesg(coord, elist, bag) };
11     out_prop ←TTCB_propose(eid, elist, tstart, TBA_MA-
       JORITY, Hash(value));
12     repeat
13       out_dec ←TTCB_decide(out_prop.tag);
14     until (out_dec.error ≠ TBA_RUNNING);
15     tstart ←tstart + T * func(α, round);
16     round ←round+1;
17     if (2f + 1 processes proposed) and (less than f + 1 pro-
       cesses proposed the same value) then
18       phase ←2;
19   until (f + 1 processes proposed the same value);
20   hash-v ←out_dec.value;

21   when receive message M
22   bag ←bag ∪ {M};

23   when (hash-v ≠ ⊥) and (∃M∈bag : Hash(M.value) = hash-v)
24   if (phase = 2) then
25     multicast M to processes in elist except those that proposed
       Hash(M.value);
26   decide M.value;
```

message has already been received (implemented by function *nextSenderMesg()*, line 10). Basically, the process first tries to see if the message from $coord = elist[k \bmod n]$ has arrived, then it tries for $elist[(k+1) \bmod n]$, next for $elist[(k+2) \bmod n]$, and so on, until a message is found. There is the guarantee that at least one message will always exist because the initial multicast (line 6) immediately puts one message in the *bag* (we use the word 'bag' to denote an unordered set of messages without duplicates). This algorithm has the interesting characteristic that it skips processes that did not manage to send their value, allowing the consensus to finish faster.

Since the value being decided might have been proposed by a malicious process, an extra precaution has to be considered. The malicious process might have sent the value just to a sufficiently large subset of processes to ensure that a decision could be made (e.g., $f$ processes). Then, the rest of the processes would never get the decided value – they would only get the corresponding hash. To solve this problem, processes have to retransmit the value to the other processes (lines 24-25). The masks from *TTCB_decide* are used to determine which processes are these.

The correctness of the protocol is proven in Appendix A.

### *4.4 FLP Impossibility Result*

Fischer, Lynch and Paterson showed that consensus in an asynchronous system has the possibility of nontermination if a single process is allowed to crash [18]. This FLP impossibility result generated research on several techniques to circumvent it both in theoretical and in real systems, e.g., randomization [30, 4, 5], unreliable failure detectors [8] and partial synchrony [16]. The precise boundaries in terms of communication synchrony, hosts synchrony and message delivery order in which this impossibility exists were studied in [13].

The purpose of this section is to discuss why FLP does not apply to our consensus protocol. The first thing to notice is that our system is not asynchronous but a combination of asynchronous and synchronous subsystems (payload and TTCB, respectively). Therefore, the FLP result does not affect the protocol. Moreover, in the *block consensus protocol* the communication boils down to the execution of TBAs, therefore it fits in the following categories of [13]: (1) it is synchronous; (2) it can be considered to be by broadcast, in the sense that all processes receive the same values; (3) it is ordered, since the TBAs are executed sequentially; (4) the receive and send operations (decide/propose in this case) are not atomic. The paper by Dolev et al. allow us also to conclude that FLP does not apply to this protocol. In relation to the *general consensus protocol*, the same reasoning applies to the consensus about the *hash* of the proposed value. The transmission of the value through the payload network does not involve a consensus, therefore FLP does not apply, also for the same reason.

## 5 Protocol Evaluation

This section evaluates the two versions of the consensus protocol in terms of time and message complexity. Since both versions use the TBA service in their implementation, we start by giving a brief overview of the current implementation of this service.

### *5.1 TBA Service*

The TBA service is implemented inside the TTCB by an agreement protocol tolerant to crash faults and under the synchronous time model. The protocol has been described in [12, 11], but we sketch it here for the reader to have an idea of its operation and complexity.

The protocol has two layers: a reliable broadcast protocol and the TBA protocol. The reliable broadcast protocol guarantees two properties: (1) all correct (non-crashed) local TTCBs deliver the same messages; (2) if the sender is correct then the message is delivered. The TTCB control channel can lose some messages due to accidental faults (e.g., electromagnetical noise), but the probability of accidental omissions in a network in an interval of time can be measured and defined with a high probability [12, 10]. This value is usually called the *omission degree* (Od). When a process proposes before *tstart*, the value and some control information are put in a table and multicasted to all local TTCBs $Od+1$ times, in order to tolerate omissions in the control channel. These messages

also include information about the last message received from all local TTCBs. This information is used by the protocol to assess if all correct local TTCBs will deliver the message.

The TBA protocol uses the reliable broadcast protocol for communication. A local TTCB can calculate the result of a TBA if one of two conditions hold: (1) if it has the proposals from all processes in *elist*, or (2) if $t \geq tstart + T_{TBA}$, where $t$ is the current instant and $T_{TBA}$ is the maximum duration for the execution of the protocol (it can be calculated since the TTCB is synchronous). $T_{TBA}$ includes a factor with the maximum asynchronism among the local TTCB clocks, since the synchronization protocol can not reduce it to zero. Finally, when a process calls *TTCB_decide*, if one of the two conditions is satisfied the TTCB returns the result (value and masks); otherwise it returns an error.

The TBA protocol is an agreement protocol variety that runs in two rounds: one round to get the messages and another to get messages confirming the reception of the first ones. There is a known theoretical minimum of $f + 1$ rounds for a consensus to tolerate $f$ faults in a synchronous system with crash faults [24]. The TBA protocol manages to improve this bound by using the omission degree mechanism described above and by making an additional assumption: if a broadcast is received by any local TTCB other than the sender, then it is received by at least *Bd* local TTCBs [2, 11]. This broadcast degree *Bd* can easily exceed half of the nodes in a LAN.

### *5.2 Time Complexity*

The time complexity of distributed algorithms is usually evaluated in terms of number of rounds or phases. Using this method, the two versions of the protocol described take one round in the best case, i.e., in a run where no failures occur. However, since these criteria can be ambiguous, Schiper introduced the notion of *latency degree* [32]. The idea is based on a variation of Lamport's logical clocks which assigns a number to an event [22], with the following rules:

1. send/multicast and local events at a process do not change its logical clock;
2. the timestamp carried by message M is defined as $ts(M) = ts(send(M)) + 1$, where $ts(send(M))$ is the timestamp of the $send(M)$ event;
3. the timestamp of a $receive(M)$ event on a process $p$ is the maximum between $ts(M)$ and the timestamp of the event at $p$ immediately preceding the $receive(M)$ event.

The notion has to be extended for systems with a wormhole. We have to introduce new rules for the distributed wormhole services, i.e., to the services that involve communication in the control channel. A distributed wormhole service can be defined in terms of two events: $w\_send$ and $w\_receive$. The event $w\_send$ represents the moment when a process calls a service to start the communication. The event $w\_receive$ represents the moment when the process gets the result of the execution of the distributed service. In relation to the TTCB TBA service, the event $w\_send$ corresponds to a call to *TTCB_propose*; $w\_receive$ corresponds to a call to *TTCB_decide* if it returns the result of the TBA. The new set of rules is:

4. a call to a local wormhole service or a $w\_send$ event at a process do not change its logical clock value;
5. the timestamp associated to a call to a distributed wormhole service $A$ is defined as $ts(A) = ts(w\_send(A)) + 2$, where $ts(w\_send(A))$ is the largest timestamp of the $w\_send$ events performed for $A$;
6. the timestamp of a $w\_receive(A)$ event on a process $p$ is the maximum between $ts(A)$ and the timestamp of the event at $p$ immediately preceding the $w\_receive(A)$ event.

These new rules were defined considering the current implementation of the TBA protocol. The protocol consists basically in every local TTCB sending the value proposed by its local process(es) to the other local TTCBs and waiting for a message from another local TTCB confirming the reception of the same value. Applying the original rules for send and receive events (rules 1-3), we derived the rules for *TTCB_propose* and *TTCB_decide*, and extrapolated to the generic rules for $w\_send$ and $w\_receive$ (rules 4-6).

Let us now define latency degree. For an execution of a consensus algorithm $\mathcal{C}$, the *latency* of $\mathcal{C}$ is the largest timestamp of all *decide* events. The *latency degree* of $\mathcal{C}$ is the minimum possible latency of $\mathcal{C}$ over all possible executions [32].

Now we calculate the latency degree for both consensus protocols applying the rules above. The logical clocks start with 0 at every process.

- *Block consensus protocol:* (1) the TBA has $ts(A) = 2$ (rules 1, 4, 5); (2) the call to *TTCB_decide*, event $w\_receive(A)$, has a timestamp of 2 at every host (rule 6); (3) every process decides at line 11 with that logical clock value so the latency degree of the protocol is 2.
- *General consensus protocol:*
  - All correct processes with same value: (1) multicast at line 6 has $ts(M) = 1$ (rules 1, 2); (2) the TBA started at line 11 has also $ts(A) = 2$ (rules 1, 4, 5); (3) if a process receives a message, the timestamp is 1 (rule 3); (4) all processes decide with a logical clock value of 2 (rule 6), and therefore the latency degree is 2.
  - Correct processes with distinct values: (1) (2) and (3) are the same; (4) processes enter in phase 2 and execute another TBA with $ts(A1) = 4$ (rules 4, 5); (5) all processes decide with a logical clock value of 4 (rule 6), and therefore the latency degree is 4.

| Protocol | Latency degree | Requirements |
|---|---|---|
| Dwork et al. [16] | 4 | Signed messages |
| Dwork et al. [16] | 7 | – |
| Malkhi & Reiter [25] | 9 or 6 | Signed messages |
| Kihlstrom et al. [21] | 4 | Signed messages |
| *Block consensus* | 2 | TTCB |
| *General consensus* | 2 or 4 | TTCB |

**Table 1.** Latency degrees for some Byzantine-resilient consensus protocols.

Table 1 compares the latency degrees of both versions of the protocol with other asynchronous Byzantine-resilient protocols that solve similar consensus problems. Although this comparison may seem awkward or unfair, the reader should notice that comparing protocols based on different system

models is a common practice in the distributed systems literature. Just to give one among many possible examples, [16] compares consensus protocols: in synchronous vs asynchronous systems; and with fail-stop vs omission vs Byzantine faults (with and without digital signatures). We also argue that this kind of comparison is useful to compare both protocols and models, especially in a paper like this that explores a recent system model.

The table shows that our protocols have good latency degrees. The translation into execution time is far from trivial, but in our case we can say that the best case execution time of the protocols is the minimum time for executing a single TBA, which is in the order of 4 ms with the current TTCB implementation.

In the presence of process failures, both versions of the protocol also have small latency degrees because they are mostly decentralized. Block consensus continues to have a latency degree of 2, and General consensus has a latency degree of 2 in case all correct processes start with the same value, and a latency degree of 4 otherwise. The other protocols presented in Table 1 are all based on a (rotating) coordinator scheme, and therefore, their performance might be affected by the failures (e.g., the first coordinators are all malicious). For instance, the latency of the protocols by Dwork et al. [16] can be as high as $4(f+1)$ for the protocol with signed messages, and $6(f+1)+1$ for the other protocol.

### *5.3 Message Complexity*

The message complexity of a protocol is evaluated in terms of the number of transmissions in the payload channel. Both versions of the protocol have the additional cost of performing TBAs which use the control channel. Table 2 shows the total number of messages sent by our protocols in the payload channel, considering the cases when a multicast is a single message (label "multicasts"), or when it is $(n-1)$ "unicasts" (plus a local delivery) of the same message.

| **Protocol** | **Multicasts** | **Unicasts** | **TBAs** |
|---|---|---|---|
| Best case | | | |
| Block consensus | 0 | 0 | 1 |
| General consensus | $n$ | $n(n-1)$ | 1 |
| Worst case | | | |
| Block consensus | 0 | 0 | no limit |
| General consensus | $2n$ | $n(n-1)+$ $+n(n-f-1)$ | no limit |

**Table 2.** Message complexities for the consensus protocols.

## 6 Related Work

The past twenty years saw several variations of the consensus problem presented in the literature. Consensus protocols can decide on a 0 or 1 bit (binary consensus), on a value with undefined size (multi-value consensus), or on a vector with values proposed by several processes (vector consensus or interactive consistency). Several Byzantine-resilient consensus protocols, using different techniques to circumvent FLP, were proposed.

Recently several works applied the idea of Byzantine failure detectors to solve consensus [25, 21, 15, 14, 3]. All these protocols use signatures. Any process $p$ can generate a signature $S(p, v)$ that cannot be forged, but which other processes can test. Likewise, they are all based on a rotating leader/coordinator per round. Malkhi and Reiter presented a binary consensus protocol in which the leader waits for a number of proposals from the others, chooses a value to be broadcasted and then waits for enough acknowledgments to decide [25]. If the leader is suspected by the failure detector, a new one is chosen and the same procedure is applied. The same paper also described a hybrid protocol combining randomization and an unreliable failure detector. The protocol by Kihlstrom et al. also solves the same type of consensus but requires weaker communication primitives and uses a failure detector that detects more Byzantine failures, such as invalid and inconsistent messages [21].

Doudou and Schiper present a protocol for vector consensus based on a *muteness failure detector*, which detects if a process stops sending messages to another one [15]. This protocol is also based on a rotating coordinator that proposes an estimate that the others broadcast and accept, if the coordinator is not suspected. This muteness failure detector was used to solve multi-value consensus [14]. Baldoni et al. described a vector consensus protocol based on two failure detectors [3]. One failure detector detects if a process stops sending while the other detects other Byzantine behavior.

Byzantine-resilient protocols based on partial synchrony, both with and without signatures, were described by Dwork et al. [16]. The protocols are based on a rotating coordinator. Each phase has a coordinator that locks a value and tries to decide on it. The protocols manage to progress and terminate when the system becomes stable, i.e., when it starts to behave synchronously.

Other techniques were also used to circumvent FLP in Byzantine-resilient consensus protocols. Randomized / probabilistic protocols can be found in [5, 6]. More recently, the condition-based approach was introduced as another means to circumvent FLP [28, 19]. Protocols based on this approach satisfy the safety properties but termination is guaranteed only if the inputs satisfy certain conditions.

The consensus protocol presented in the paper is one of the first existing protocols based on the TTCB wormhole. Previously we designed a Byzantine-resilient reliable multicast protocol [10]. This protocol uses the TBA to multicast a reliable hash of a message. The current paper shows a different way of using the TTCB TBA service: to make a voting on the values proposed by the processes, and to decide when enough processes voted the same, or simply voted something.

## 7 Conclusion

The need for more trustworthy systems in a widely connected world is raising an increasing interest in the development of practical Byzantine-resilient protocols and applications. In this

context, we are exploring a secure and real-time wormhole – the TTCB – to support the execution of this type of protocols.

The objective of the current paper is twofold: (1) to show the power of the wormhole model; and (2) to show how to develop novel algorithmic solutions in the model. These goals are pursuit by presenting a consensus protocol. Although this protocol may seem simple, it requires a new algorithmic perspective, since it is based on a dual system, both in terms of time and security. We are also not aware of any consensus protocol executed with the assistance of a "low-level" simple agreement service. The protocol has low time and message complexities.

*Acknowledgements.* We warmly thank Danny Dolev, Rachid Guerraoui, André Schiper, the anonymous reviewers and the handling editor, Dahlia Malkhi, for their comments that greatly assisted us in improving the paper.

## References

1. A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Veríssimo, M. Waidner, and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*. January 2002. http://www.research.ec.org/maftia/deliverables/D21.pdf.
2. Ö. Babaoğlu, R. Drummond, and P. Stephenson. The impact of communication network properties on reliable broadcast protocols. In *Proceedings of the 16th IEEE International Symposium on Fault-Tolerant Computing*, pages 212–217, July 1986.
3. R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, pages 1–16, June 2000.
4. M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.
5. G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.
6. C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132, July 2000.
7. A. Casimiro, P. Martins, and P. Veríssimo. How to build a Timely Computing Base using Real-Time Linux. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 127–134, September 2000.
8. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
9. P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper. In *Real-Time Linux Workshop*, November 2000.
10. 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.
11. M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel (extended version). DI/FCUL TR 01–12, Department of Computer Science, University of Lisbon, 2001.
12. 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.
13. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
14. A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash-stop to Byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50, May 2002.
15. A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes. Technical Report 97/30, EPFL, 1997.
16. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
17. M. J. Fischer. The consensus problem in unreliable distributed systems (A brief survey). In M. Karpinsky, editor, *Foundations of Computing Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer-Verlag, 1983.
18. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
19. R. Friedman, A. Mostefaoui, S. Rajsbaum, and M. Raynal. Distributed agreement and its relation with error-correcting codes. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 63–87, October 2002.
20. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.
21. K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, January 2003.
22. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
23. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
24. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
25. D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.
26. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
27. F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, pages 214–222, July 1987.
28. A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. In *Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 152–162, July 2001.
29. D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer-Verlag, November 1991.
30. M. O. Rabin. Randomized Byzantine Generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, November 1983.
31. R. Reischuck. A new solution for the Byzantine general's problem. Technical Report RJ 3673, IBM Research Lab., November 1982.

32. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10:149–157, October 1997.
33. B. Tobotras. Linux Capabilities FAQ 0.2. ftp://ftp.guardian.no/pub/free/linux/capabilities/capfaq.txt, 1999.
34. 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.
35. P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, August 2002.
36. 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.
37. P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global clock service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, May 1997.

## A Correctness Proofs

This section proves that Protocols 1 and 2 solve consensus as defined by the properties of Validity, Agreement and Termination in Section 4.1, provided that at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes fail. We assume the system model in Section 2 and the weak synchrony assumption in Section 2.2. We assume each process successfully called the Local Authentication service and established a secure channel with its local TTCB before the execution of the protocols (Section 3). If an attacker manages to disclose the pair $(eid, key)$ established by this service, the secure channel is no longer secure so we considered the process to be failed. We assume the TBA service satisfies its specification in terms of properties TBA1 to TBA5 in Section 3.1.

### *A.1 Block Consensus Correctness Proof*

**Theorem 1.** *If all correct processes propose the same value* v*, then any correct process that decides, decides* v *(Validity).*

*Proof.* The theorem applies only if all correct processes propose the same value *v*. There are at least $2f + 1$ correct processes since we assume $f \leq \lfloor \frac{n-1}{3} \rfloor$. The algorithm is basically a loop inside lines 3 to 10. All correct processes begin with the same *tstart* that works as the loop counter.

Each round of the loop, all correct processes call *TTCB_propose* and get the result of the TBA *out_dec* calling *TTCB_decide* (line 6). *out_dec* contains the (or one of the) value(s) proposed by *more* processes before *tstart* (due to property TBA4, with the decision function *TBA_MAJORITY*) and the two masks saying which processes proposed the value decided and which proposed any value before *tstart*. Each round can satisfy one of two cases, depending on the number of processes $k$ that proposed before *tstart*:

*Case 1.* ($k < 2f + 1$): This case can be subdivided in another two. (Case 1a): If no $f + 1$ processes proposed the value decided, then the loop goes to the next round (line 10). (Case 1b): If $f + 1$ processes proposed the value decided then this value is necessarily *v*, since there are at most $f$ failed processes (the theorem assumes all correct processes propose *v*). In the end of the round, the loop terminates since $f + 1$ proposed the same value (line 10). The value *v* is decided (line 11).

*Case 2.* ($k \geq 2f + 1$): In this case, at least $f + 1$ of the processes that proposed are correct and they are the majority, since at most $f$ can be failed. Therefore, the value decided by the TBA is *v* (line 6), the loop terminates (line 10) and *v* is decided by the protocol (line 11).

Any correct process that decides, decides in cases (1b) or (2), therefore it decides *v*. □

**Theorem 2.** *No two correct processes decide differently (Agreement).*

*Proof.* Two correct processes execute the same TBAs, since they start with the same *tstart* (Section 1) and TBA returns the same values to all processes (property TBA3). Two correct processes exit the loop in the same round since they test the same condition (line 10) with the same results of TBA's. They return the same result for the same reason (line 11). □

**Theorem 3.** *Every correct process eventually decides (Termination).*

*Proof.* The synchrony assumption in Section 2.2 states that there is an unknown *processors stabilization time* (PST) such that the processing delays are bounded from time PST onward. Therefore, eventually there is a round when at least $2f + 1$ processes manage to call *TTCB_propose* before the *tstart* $>$ *PST* deadline. When that happens all correct processes of that subset with at least $2f + 1$ eventually decide (lines 5-11, given properties TBA1 and TBA5). There may exist $f$ correct processes which did not manage to call *TTCB_propose* before that *tstart*. However, they will make that call later, get the result of the TBA (line 6) and terminate (lines 10-11). □

### *A.2 General Consensus Correctness Proof*

**Lemma 1.** *If all correct processes propose the same value then the protocol does not change to phase 2.*

*Proof.* The change to phase 2 is tested in line 17. If $2f + 1$ processes proposed then at least $f + 1$ of them are correct. Since we are considering that all correct processes proposed the same value, the second part of the condition is not satisfied. Therefore, if the first part of the condition in line 17 is satisfied, the second is not, and the protocol does not change to phase 2. □

**Theorem 4.** *If all correct processes propose the same value* v*, then any correct process that decides, decides* v *(Validity).*

*Proof.* The theorem applies only when all correct processes propose the same value *v*, therefore the protocol does not change to phase 2 (Lemma 1). The phase 1 of the protocol is very similar to the Block Consensus protocol, therefore the

proof that any correct process that decides, decides the same hash $H(v)$ follows from the proof of Theorem 1. If a process is correct then it eventually receives its own message with *v* (lines 6, 21). Therefore, any correct process that decides, decides *v* (lines 23, 26). □

**Theorem 5.** *No two correct processes decide differently (Agreement).*

*Proof.* The proof that no two correct processes decide different hashes is similar to Theorem 2. If two correct processes decide the same hash then they decide the same value due to the properties assumed for the hash function (lines 23 and 26, Section 4.3). □

**Theorem 6.** *Every correct process eventually decides (Termination).*

*Proof.* The proof that either all correct processes eventually terminate in phase 1 (line 19) or they change to phase 2 (line 17) is similar to the proof of Theorem 3.

Let us now prove that all correct processes in phase 2 eventually decide. All correct processes multicast their values $v_i$ to all others (line 6). Attending to the communication model, eventually every correct process receives the messages with the values $v_i$ from all correct processes. Line 10 chooses the value $v_j$ proposed by the process with index $(r \mod n)$ in *elist* or the next one available. Again using the reasoning of the proof of Theorem 3, eventually $f + 1$ processes manage to propose the same $H(v_j)$, which is decided by the TBA. If a process has the value $v_j$ in $bag$ then it decides immediately (lines 19-20, 23-26). If a process $p$ does not have the value $v_j$ then it will eventually receive it, since at least one other correct process has $v_j$ ($f + 1$ processes have it) and multicasts it (line 24-25). After receiving $v_j$, $p$ decides it (lines 21-26). □

**M** iguel Correia is an assistant professor of the Department of Informatics, University of Lisboa Faculty of Sciences. He received a PhD in Computer Science at the University of Lisboa in 2003. Miguel Correia is a member of the LASIGE laboratory and the Navigators research group. He was involved in the EC-IST MAFTIA project and the FCT DeFeATS project, both in the area of fault and intrusion-tolerance. More information about him is available at http://www.di.fc.ul.pt/∼mpc

**N** uno Ferreira Neves has been an assistant professor of the Department of Informatics, University of Lisboa since 1998. He received a Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign. His research interests are in parallel and distributed systems, in particular in the areas of security and fault-tolerance. His work has been recognized with a Fulbright Fellowship during the doctoral studies and with the William C. Carter Best Student Paper award at the 1998 IEEE International Fault-Tolerant Computing Symposium. More information about him is available at http://www.di.fc.ul.pt/∼nuno

**L** au Cheuk Lung is a associate professor in the Department of Computer Science at Pontifical Catholic University of Paraná - Brazil, where he has been working since 2003. Currently, he is conducting research in fault tolerance, security in distributed systems and middleware. From 1997 to 1998, he was a associate reseach fellow at University of Texas at Austin, working on the Nile Project. From 2001 to 2002, he was a postdoctoral research associate in the Computer Science Department at University of Lisbon, Portugal. In 2001, Lau received a PhD from Santa Catarina Federal University - Brazil.

**P** aulo Veríssimo is professor of the Department of Informatics, University of Lisboa Faculty of Sciences (http://www.di.fc.ul.pt/∼pjv). He is coordinator of the CORTEX IST/FET project and belongs to the Executive Board of the "CaberNet European Network of Excellence". He is Chair of the IEEE Technical Committee on Fault Tolerant Computing. Paulo Veríssimo leads the Navigators research group of LASIGE, and is currently interested in: architecture, middleware and protocols for dependable distributed systems, namely the facets of adaptive real-time and fault/intrusion tolerance. He is author of more than 100 refereed publications in international scientific conferences and journals in the area, he is co-author of four books.