Byzantine Consensus in Asynchronous Message-Passing Systems: a Survey*

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

1 Introduction

This section presents a short survey on Byzantine consensus – or Byzantine agreement – in asynchronous message-passing distributed systems. Consensus is a classical distributed systems problem, first introduced in [63], with both theoretical and practical interest [51, 37]. The problem can be stated informally as: how to ensure that a set of distributed processes achieve agreement on a value despite a number of faulty processes. The importance of this problem derives from several other distributed systems problems being reducible or equivalent to it. Examples are atomic broadcast [40, 16, 21], non-blocking atomic commit [39], group membership [39], and state machine replication [68].

These relations between consensus and other distributed problems are important because many results stated for consensus automatically apply to these other problems. Among these results, the most relevant is probably the impossibility of solving consensus deterministically in an asynchronous system if a single process can crash, often called the FLP result [32]. This result lead to a large number of works that try to circumvent it, i.e., to slightly modify the system model in such a way that consensus becomes solvable. Examples include randomization [66, 7], failure detectors [16, 53], partial-synchrony [30, 26], and wormholes [19, 62].

There is another set of results that have to do with how many faulty processes can be tolerated and how efficiently the problem can be solved, i.e., about the performance of algorithms that solve consensus. Important metrics of efficiency are the minimum number of asynchronous steps and messages needed to solve consensus (a survey of early results is given in [50]). These metrics are important to assess the scalability of the algorithms, although recent work has shown that they can hide as much as they show [41, 55].

Algorithms that solve consensus vary heavily depending on the system model, like algorithms that solve any other distributed systems problem. This section considers only message-passing algorithms that tolerate Byzantine (i.e., arbitrary) faults in asynchronous systems (i.e., without time assumptions).

*Appeared in "Resilience-building Technologies: State of Knowledge", RESIST Network of Excellence deliverable D12, Part Algo, Chapter 1, September 2006.

The reason for choosing this system model is not merely its theoretical interest. Today, algorithms based on this model are being used as important building blocks for the construction of secure applications based on a recent approach: *intrusion tolerance* [33, 1, 44, 73, 5]. This approach can be considered to be part of the ongoing effort to make computing systems, Internet included, more secure vis-a-vis the large number of security incidents constantly reported by entities like the CERT/CC[1].

Some brief comments are due on the three main aspects of the system model we consider: message-passing, Byzantine faults and asynchrony.

The message-passing model is the natural choice for algorithms supposed to be executed not in parallel machines but generic distributed systems, like those built on top of the Internet. An alternative system model is shared-memory [4, 34, 52, 3, 9], but in the kind of system we are considering the shared memory itself has to be implemented using message passing algorithms.

Arbitrary faults, usually called Byzantine after the paper by Lamport et al. [47], do not put any constraint on how processes fail. This sort of assumption or, better said, of non-assumption about how processes fail, is specially adequate for systems where malicious attacks and intrusions can occur. For instance, a process might be executed by an attacker that modifies the behavior of its implementation of the algorithm arbitrarily. Interestingly, assuming Byzantine faults, instead of the more typical assumption of crash faults, leads to more complex and challenging algorithms.

Asynchrony might also be better described as a non-assumption about time properties. This (non-) assumption is important because attackers can often violate some time properties by launching denial-of-service attacks against processes or communications. For instance, the attacker might delay the communication of a process indefinitely, breaking any assumptions about process timeliness.

This system model – assuming Byzantine faults and asynchrony – is very generic. Algorithms in this model have to deal with two independent degrees of uncertainty: in terms of faults and time. This leads to algorithms that besides being able to deal with malicious behavior, have the virtue of also running correctly in more benign environments, like those in which only crash faults occur or in which delay bounds are attained.

The consensus problem is defined for a set of n known processes. Currently there is a trend of research on large dynamic systems in which the number of involved processes is unknown [59, 2]. Consensus, however, is still not defined in this context.

2 Byzantine Consensus Definitions

This section defines the consensus problem and several of the variations considered in the literature. We say that a process is *correct* if it follows its algorithm until completion, otherwise it is said to be *faulty*.

A *binary consensus* algorithm aims to achieve consensus on a binary value $v \in \{0,1\}$. Each process proposes its *initial value* (binary) and decides on a value v. The problem can be formally defined in terms of three properties:

- *Validity:* If all correct processes propose the same value v, then any correct process that decides,

[1] http://www.cert.org/stats

decides v.

- *Agreement:* No two correct processes decide differently.
- *Termination:* Every correct process eventually decides.

The first 2 properties are *safety* properties, i.e., properties that say that some bad thing cannot happen, while the last is a *liveness* property, i.e., a property that states good things that must happen.

Multi-valued consensus is apparently similar to binary consensus, except that the set of values has arbitrary size, i.e., $v \in V$ and $|V| > 2$. Multi-valued consensus algorithms have been studied in the literature using several Validity properties, while the Agreement and Termination properties remained essentially the same (with minor exceptions for Termination that we will see later). Some papers use the following Validity property [30, 53, 43]:

- *Validity 1:* If all correct processes propose the same value v, then any correct process that decides, decides v.

Others use the following [28, 27, 6]:

- *Validity 2:* If a correct process decides v, then v was proposed by some process.

Both properties are somewhat weak. Validity 1 does not say anything about what is decided when the correct processes do not propose all the same v, while Validity 2 does not say anything about what is the value decided (e.g., is it the value proposed by the correct processes if all of them propose the same?). Recently, a definition that gives more detail about what is decided has been proposed [21]. The definition has three Validity properties:

- *Validity 1:* If all correct processes propose the same value v, then any correct process that decides, decides v.
- *Validity 2':* If a correct process decides v, then v was proposed by some process or $v = \perp$.
- *Validity 3:* If a value v is proposed only by corrupt processes, then no correct process decides v.

The first two are essentially the Validity properties already introduced, except that Validity 2' allows the decision of a value $\perp \notin V$. The third property is inspired by the original definition in the context of the "Byzantine Generals" metaphor used in the classical paper by Lamport et al. [47]. The definition was "(1) All loyal generals decide upon the same plan of action; (2) A small number of traitors cannot cause the loyal generals to adopt a bad plan.". That paper however, considered a synchronous system, i.e., a system in which there are known delay bounds for processing and communication.

This concern about the practical interest of multi-valued consensus defined in terms of Validity 1 or Validity 2 lead also to the definition of *vector consensus* [28]. The difference in relation to the previous versions of the problem is once again the Validity property. The decision is no longer a single value but a vector with some of the initial values of the processes, at least $f + 1$ of which are from correct processes. The validity property is now stated as:

- *Vector validity:* Every correct process that decides, decides on a vector V of size n:
  - $\forall_{p_i}$: if p_i is correct, then either *V[i]* is the value proposed by p_i or $\perp$;
  - at least $(f+1)$ elements of V were proposed by correct processes.

Vector consensus is a variation for asynchronous systems of the classical *interactive consistency* problem [63]. Interactive consistency is a consensus on a vector with values from all correct processes. However, in asynchronous systems a correct process can be very slow so it is not possible to guarantee that values from all correct processes are obtained and still ensure Termination. Therefore, vector consensus ensures only that $f+1$ of the values in the vector are from correct processes. This is clearly more interesting than multi-valued consensus since it tells much more about the initial values of the correct processes. Interestingly, vector consensus has been proved to be equivalent to multi-valued consensus defined with the validity properties Validity 1, Validity 2' and Validity 3 [21].

Some other variations of consensus have been studied in the literature. For instance, the *k-set consensus* problem in which the correct processes can decide at most k different values [17, 24].

A somewhat different kind of definition is the one used in the Paxos algorithms [45, 46, 48, 74, 54]. The idea is that processes play one or more of the following roles: *proposers* (propose values), *acceptors* (choose the value to be decided) and *learners* (learn the chosen value). The problem can be defined in terms of five properties [46, 54]:

- Only a value that has been proposed may be chosen.
- Only a single value may be chosen.
- Only a chosen value may be learned by a correct learner.
- Some proposed value is eventually chosen.
- Once a value is chosen, correct learners eventually learn it.

The first three properties are safety properties, while the last two are liveness properties. This definition is interesting because it allows a simple implementation of state machine replication in the crash failure model [68, 46]. However, in the Byzantine failure model this is not so simple [14, 54].

3 FLP Impossibility

The most cited paper about consensus is probably the one that proves the impossibility of solving consensus deterministically in an asynchronous system if a single process can crash [32]. This result is often called the FLP impossibility result, after its authors' names, Fischer, Lynch and Paterson. The consensus definition used to prove the result is even weaker than the first definition in Section 2: validity is more relaxed and termination is required for a single process.

The idea is that the uncertainty in terms of timeliness (asynchrony) combined with the uncertainty in terms of failure (even if failures are only crashes and only one process can fail) does not allow

any deterministic algorithm to guarantee the binary consensus definition given in the previous section. More precisely, the reason for the impossibility is that in an asynchronous system it is impossible to differentiate a crashed process from another that is simply slow (or connected by a slow network link). In the years that followed the statement and proof of this result, a few alternative proofs have been given (a discussion of these proofs can be found in [50]).

The FLP result says when consensus is not solvable. However, from a practical point of view it is more important to know when it can be solved. A first detailed study of this issue was presented by Dolev, Dwork and Stockmeyer for crash faults [26]. The paper identified five relevant parameters that affect solvability: synchrony/asynchrony of the processes; synchrony/asynchrony of the communication; ordered/unordered message delivery; broadcast/point-to-point communication; and atomic/not-atomic receive and send. This lead to 32 different models. The paper showed that different degrees of synchronism allow deterministic algorithms to tolerate different numbers of crash faults (there was no study for Byzantine faults).

To solve consensus, an algorithm has to *circumvent* the FLP impossibility result. This word, circumvent, is quite unprecise so it is important to discuss its meaning. The idea is to slightly modify either the system model or the problem definition considered in [32] to allow the problem to be solvable. These modifications change the conditions in which FLP was proven so, to be precise, the result simply no longer applies. However, researchers in the area prefer to call it "circumventing" the result, to pass the idea that the conditions are close to those in which the result applies.

An observation about FLP with interesting practical consequences is that if we discard one of the properties that define consensus, we can enforce the two others. This observation lead researchers to design consensus algorithms in the following way:

- the algorithm solves consensus if the technique used to circumvent FLP works as it is assumed to;
- the algorithm satisfies the safety properties even if the technique used to circumvent FLP does not work as it assumed to [36, 38].

The idea is that if something bad happens, like an additional time assumption not being satisfied, the algorithm may not terminate, but Validity and Agreement properties will always be satisfied. This notion has been recently called *indulgence* in the context of system models augmented with eventual failure detectors [36, 38], but almost all consensus algorithms satisfy it. The only exception that we are aware of is the randomized algorithm in [13], which always terminates but only satisfies Agreement with a certain probability.

There are several ways to look into the techniques to circumvent FLP. We use a classification in four types of techniques, which we present in more detail in the following section:

- techniques that sacrifice determinism, leading to probabilistic algorithms;
- techniques that add time to the model;
- techniques that enrich the system model with an oracle;
- techniques that enrich the problem definition.

4 Circumventing FLP

The following sections introduce the techniques to circumvent FLP and algorithms that use them.

4.1 Sacrificing Determinism

The FLP impossibility result applies to *deterministic* algorithms so a solution to circumvent it is by using *randomization* to design *probabilistic* algorithms. More specifically, the idea is to substitute one of the properties that define consensus by a similar property that is satisfied only with a certain probability. For the reasons mentioned above, almost all algorithms choose to modify the Termination property, which becomes:

- *P-Termination:* Every correct process eventually decides with probability 1.

The single exception that we are aware of, already mentioned above, does not modify Termination but Agreement, so agreement on the value decided is reached with a certain probability [13].

Randomized Byzantine consensus algorithms have been around since Ben-Or's and Rabin's seminal papers [7, 66]. Virtually all randomized consensus algorithms are based on a random operation, *tossing a coin*, which returns values 0 or 1 with equal probability.

These algorithms can be divided in two classes depending on how the tossing operation is performed: there are those that use a *local coin* mechanism in each process (starting with Ben-Or's work [7]), and those based on a *shared coin* that gives the same values to all processes (initiated with Rabin's work [66]).

Typically, local coin algorithms are simpler but terminate in an expected exponential number of communication steps [7, 10], while shared coin algorithms require an additional coin sharing scheme but can terminate in an expected constant number of steps [66, 69, 8, 13, 12, 35]. The original Rabin algorithm required a trusted dealer to distribute key shares before the execution of the algorithm [66]. This unpractical operation is no longer needed in more recent algorithms [13, 12].

Randomized consensus algorithms have often been assumed to be inefficient due to their high expected message and time complexities, so they have remained largely overlooked as a valid solution for the deployment of fault-tolerant distributed systems. Nevertheless, two important points have been chronically ignored. First, consensus algorithms are not usually executed in oblivion, they are run in the context of a higher-level problem (e.g., atomic broadcast) that can provide a friendly environment for the "lucky" event needed for faster termination (e.g., many processes proposing the same value can lead to a quick termination). Second, for the sake of theoretical interest, the proposed adversary models usually assume a strong adversary that completely controls the scheduling of the network and decides which processes receive which messages and in what order. In practice, a real adversary usually does not possess this ability, but if it does, it will probably perform simpler attacks like blocking the communication entirely. Therefore, in practice, the network scheduling can be "nice" and lead to a speedy termination. A recent paper shows that this is true and that these algorithms can be practical [55].

4.2 Adding Time to the Model

The notion of *partial synchrony* was introduced by Dwork, Lynch and Stockmeyer in [30]. A partial synchrony model captures the intuition that systems can behave asynchronously (i.e., with variable/unkown processing/communication delays) for some time, but that they eventually stabilize and start to behave (more) synchronously. Therefore, the idea is to let the system be mostly asynchronous but to make assumptions about time properties that are eventually satisfied. Algorithms based on this model are typically guaranteed to terminate only when these time properties are satisfied.

Dwork et al. introduced two partial synchrony models, each one extending the asynchronous model with a time property:

- *M1:* For each execution, there is an unknown bound on the message delivery time Δ, which is always satisfied.
- *M2:* For each execution, there is an unknown global stabilization time GST, such that a known bound on the message delivery time Δ is always satisfied from GST onward.

Chandra and Toueg proposed a third model, which is similar but weaker [16]:

- *M3:* For each execution, there is an unknown global stabilization time GST, such that an unknown bound on the message delivery time Δ is always satisfied from GST onward.

Two Byzantine consensus algorithms, one based on M1 and the other on M2, are presented in the original paper by Dwork et al. [30]. The algorithms are based on a rotating coordinator. Each phase has a coordinator that locks a value and tries to decide on it. The algorithms manage to progress and terminate when the system becomes stable, i.e., when it starts to behave synchronously. There is still no algorithm or proof that M1 allows Byzantine consensus to be solved, although it has been shown to be enough to solve crash-tolerant consensus [16].

The timed asynchronous model enriches the asynchronous system model with hardware clocks that can be used to detect the violation of time bounds [23]. Cristian and Fetzer have shown that it is possible to solve consensus in this model, although the problem of Byzantine consensus has not been studied [31].

4.3 Augmenting the System Model with an Oracle

The section describes how FLP can be circumvented using oracles. The original idea of circumventing FLP using oracles was introduced by Chandra and Toueg [16]. The oracle in that case is a *failure detector*, i.e., a component that gives hints about which processes are failed or not failed. Remember that FLP derives from the impossibility of distinguishing if a process is faulty or simply very slow. Therefore, intuitively, having a hint about the failure/crash of a process may be enough to circumvent FLP. Notice however that augmenting the system model with a failure detector is equivalent to modifying the time model since (useful) failure detectors cannot be implemented in asynchronous systems. In fact, time assumptions, like those made in partial synchrony models, are usually necessary. The single exception

that we are aware of is the requirement for some order pattern in the messages exchanged by the failure models in the solution presented in [56].

The following section presents failure detectors and the next one wormholes, which are a more generic concept. Other types of oracles have been presented in the literature, but they have not been used with Byzantine faults. Examples include the Ω detector, which provides hints about who is the leader process [15], and ordering oracles, which provide hints about the order of messages broadcasted [64].

4.3.1 Failure Detectors

The original idea of failure detectors was to detect or, more precisely, to suspect the crash of a process. Each process has attached a failure detector module and the set of all these modules formed the failure detector.

Recently, several works applied the idea of Byzantine failure detectors to solve consensus [53, 43, 28, 6, 27, 35]. The main differences in relation to crash failure detectors is that (1) Byzantine failure detectors can neither be made completely independent of the algorithm in which they are used [27], nor (2) detect all Byzantine faults, only certain subsets [43].

Malkhi and Reiter presented a binary consensus algorithm 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 [53]. 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 algorithm combining randomization and an unreliable failure detector. The algorithm 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 [43].

Doudou and Schiper presented an algorithm for vector consensus based on a *muteness failure detector*, which detects if a process stops sending messages to another one [28]. This algorithm 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 [27]. Another efficient algorithm based on a muteness failure detector was presented by Friedman et al. [35].

Baldoni et al. described a vector consensus algorithm based on two failure detectors [6]. One failure detector detects if a process stops sending messages (muteness) while the other detects other Byzantine failures. This latter detector is implemented using an interesting solution based on a finite-state automaton that monitors the behavior of the algorithm.

All algorithms based on failure detectors that we are aware of are indulgent, i.e., they satisfy the safety properties of consensus (Validity and Agreement) even if the failure detector does not behave "nicely". Examples of undesirable behavior of a failure detector are not detecting a subset of Byzantine behavior or the muteness of a process.

4.3.2 Wormholes

Wormholes are an extension to a system model with stronger properties than the rest of the system. Wormholes are materialized as enhanced components that provide processes with a means to obtain a few simple privileged functions with "good" properties otherwise not guaranteed by the normal environment [71, 72]. For example, a wormhole can provide timely or secure functions in, respectively, asynchronous or Byzantine systems. This contrasts with work on failure detectors, which tries to abstract the minimum requirements on hints about failures to solve consensus. The idea is more generic and has to do with what are the distributed system models that allow to have desirable levels of predictability in systems that are mostly uncertain in terms of properties like time and security [72].

Wormholes are closely related to the notion of *architectural hybridization*, a well-founded way to substantiate the provisioning of those "good" properties on "weak" environments. In the case that we are interested in here, we assume that our system is essentially asynchronous and Byzantine, so when implementing the model we should not simply postulate that parts of it behave in a timely or secure fashion, or these assumptions might naturally fail. Instead, those parts should be built in a way that our claim is guaranteed with high confidence.

The first paper that presented a consensus algorithm based on a wormhole [19] used a specific wormhole, a device called *Trusted Timely Computing Base* (TTCB) [22]. Technically, the TTCB is a secure real-time and fail-silent distributed component. Applications implementing the consensus algorithm run in the normal system, i.e., in the asynchronous Byzantine system. However, the TTCB is locally accessible to any process, and at certain points of the algorithm the processes can use it to execute correctly (small) crucial steps. The consensus algorithm relies mostly on a TTCB service called Trusted Block Agreement Service, which essentially makes an agreement on small values proposed by a set of processes. The idea is to use this service to make agreement on the hash of the value proposed by the majority of the processes. Later, a simpler multi-valued consensus algorithm and a vector consensus based on wormholes were also proposed [61, 62].

4.4 Modifying the Problem

This section describes how FLP can be circumvented by weakening the definition of consensus. Currently, we are aware of a single type of algorithm that fits in this category: algorithms based on the condition based approach. These algorithms terminate if the initial values of the processes satisfy certain conditions, but satisfy the safety properties – Validity and Agreement – even if the conditions are not satisfied.

Let us define the *input vector* for an execution of a consensus algorithm as the vector I in which each $I[i]$ is the initial value of process p_i. The condition based approach identifies sets of input vectors for which the consensus algorithm terminates (besides satisfying Validity and Agreement) [57, 58, 34]. Conditions on input vectors have been shown to be directly related to error correcting codes. In fact, crash failures correspond to erasure errors in the context of error correcting codes, while Byzantine failures correspond to corruption errors [34].

An argument in favor of this sort of trade-off between Termination and conditions on input vectors

is made in [34]. A first reason is that it makes sense to use the approach to efficiently solve consensus problems in which the initial values really satisfy the conditions, but to guarantee safety even if this assumption does not hold. A second reason is that the conditions can serve as a guideline that allows the designer to augment the system (modifying the system model) with the minimum synchrony needed to ensure the solvability of the problem.

The single paper about the condition based approach that we are aware of that deals with Byzantine failures is [34]. This paper presents simple algorithms to solve multi-valued and k-set consensus.

5 Performance and Scalability

Byzantine distributed algorithms have been evaluated using several different metrics. Ultimately, the objectives are to understand how an algorithm works and how it behaves in practice:

- How will it *perform*? Or, more precisely, what will be its latency (time needed to run) and throughput (number of executions per unit of time)?
- How will it *scale*, i.e., what is the relation between its performance and the number of processes executing it?
- What will be its *resilience*, i.e., how many faulty processes will it tolerate?

The first two parameters are usually evaluated theoretically in terms of time, message and communication complexities. In asynchronous systems, time complexity is usually measured in terms of the maximum number of *asynchronous steps*. An asynchronous step involves a process sending a message and receiving one or more messages sent by the other processes. The message complexity is measured by the *number of messages sent* and the communication complexity by the *number of bits sent*. Cryptographic operations often have some impact in the processing time, especially public-key cryptography operations, so the evaluation should also take into account, e.g., the number of signatures made and evaluated. Recently it has been shown that the minimum number of asynchronous steps for Paxos consensus is 2 [29, 54].

These metrics are not so simple to assess as it may seem, since they usually depend on the occurrence of faults. Therefore, the evaluations should consider at least two cases: failure-free executions and executions in which the maximum number of processes ($\lfloor (n-1)/3 \rfloor$) is faulty (Byzantine). Other aspects, like the correct processes having the same initial value, can influence the performance evaluation and should also be taken into account.

For probabilistic algorithms, these parameters can only be stated probabilistically so usually the metrics considered are the *expected* number of asynchronous steps, messages sent, bits sent. The literature usually assesses these values in the worst case, i.e., most unfavorable combination of initial values, failures and network scheduling of the messages.

Despite the importance of these theoretical metrics, it has been argued that they may not reflect correctly the behavior of the algorithms in practice [41]. A recent paper has shown that this is true and

that, for instance, randomized binary consensus algorithms that in theory run in high numbers of steps, in practice may execute in only a few communication steps under realistic conditions [55].

The third parameter above, resilience, is probably the simplest since it can be assessed precisely for an algorithm. The optimal resilience for Byzantine consensus in all system models that we are aware of is $n/3$, i.e., less than $n/3$ out of n processes can fail for the algorithm to run correctly [47, 10, 30, 21].

In relation to resilience, it is important to note that there is no point in making assumptions about the maximum number of processes that can be faulty if there are common modes of failure [65]. For Byzantine failure model, common modes of failure are caused by identical bugs or vulnerabilities in all (or several) processes [73]. Independence of failure of processes can be enforced by using diversity of design [25, 49].

6 Related and Equivalent Problems

In the introduction, we mentioned that there are several distributed systems problems equivalent to consensus. In this section we give more details about this issue.

Given two distributed problems A and B, a *transformation* from A to B is an algorithm that converts any algorithm that solves A into an algorithm that solves B [40]. Problems A and B are said to be *equivalent* if there is a transformation from A to B and a transformation from B to A. Sometimes the equivalence is not generic but assumes some specificity of the system model, like the existence of signatures.

The first equivalences and transformations were established for the crash failure model. In this model, multi-valued consensus has been proved to be equivalent to atomic (or total order) broadcast [40, 16]. Transformations from consensus to several problems have been also presented: non-blocking atomic commit [39], group membership [39], and state machine replication [68]. Only some of these equivalences/transformations extend to the Byzantine failure model. For instance, non-blocking atomic commit commits a transaction if all resources say 'commit' and aborts it one or more say 'abort'. With Byzantine failure model, a faulty process can simply abort all transactions preventing the system from working as expected, so clearly there is no transformation from consensus to non-blocking atomic commit.

The equivalence of (Byzantine) atomic broadcast and consensus has been first proved for systems with signatures in [11]. A similar result but without the requirement of signatures has been proved in [21]. Both proofs are independent of the technique used to circumvent FLP. Atomic broadcast, or total order broadcast, is the problem of delivering the same messages in the same order to all processes.

We are not aware of other transformations from Byzantine consensus to other distributed systems problems. However, there is probably a transformation from vector consensus, which has been shown to be equivalent to a variation of multi-valued consensus [21], to group membership. A group membership algorithm makes agreement about a sequence of views, which are numbered events with the identifiers of the members of a group of processes (see, e.g., the survey in [18]). The view can be modified by events like the addition of members to a group, the removal of failed members, and the removal of members by their own initiative. The Byzantine-resilient membership algorithms available give this intuition that a

transformation might be defined [67, 42, 20].

Several transformations from a variation of (Byzantine) consensus to another have been presented in the literature. Turpin and Coan presented a transformation from binary to multi-valued consensus for synchronous systems [70]. Toueg and Cachin et al. presented similar transformations for asynchronous systems, both requiring signatures [69, 11]. Transformations from binary to multi-valued consensus, and from multi-valued to vector consensus, without signatures, were presented in [21].

7 Conclusion

Consensus is an important problem in distributed systems since it can be considered to be the "greatest common subproblem" of several other problems [60]. This section presents a short survey about research on consensus in asynchronous message-passing systems prone to Byzantine faults. Algorithms that solve the several variations of this problem and the equivalent problem of atomic broadcast are currently being used as fundamental building blocks in secure and intrusion-tolerant applications. Therefore, the importance of the consensus problem is undeniable.

8 Acknowledgments

We thank Michel Raynal and David Powell for their comments on preliminary drafts of this survey that greatly assisted us in improving it.

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*. Jan. 2002.

[2] M. Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, 35(2):36–59, 2004.

[3] N. Alon, M. Merrit, O. Reingold, G. Taubenfeld, and R. Wright. Tight bounds for shared memory systems acessed by Byzantine processes. *Distributed Computing*, 18(2):99–109, 2005.

[4] P. C. Attie. Wait-free Byzantine consensus. *Information Processing Letters*, 83(4):221–227, Aug. 2002.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan.-Mar. 2004.

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

[7] 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, Aug. 1983.

[8] M. Ben-Or. Fast asynchronous Byzantine agreement. In *Proceedings of the 4th ACM Symp. on Principles of Distributed Computing*, pages 149–151, Aug. 1985.

[9] A. N. Bessani, M. Correia, J. S. Fraga, and L. C. Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, July 2006.

[10] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, Aug. 1984.

[11] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In J. Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer-Verlag, 2001.

[12] 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.

[13] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 42–51, 1993.

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

[15] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

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

[17] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

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

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

[20] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Worm-IT – a wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, 2006. to appear.

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

[22] 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, Oct. 2002.

[23] F. Cristian and C. Fetzer. The timed asynchronous system model. In *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 140–149, 1998.

[24] R. de Prisco, D. Malki, and M. Reiter. On k-set consensus problems in asynchronous systems. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 257–265, May 1999.

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

[26] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987.

[27] 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.

[28] A. Doudou and A. Schiper. Muteness detectors for consensus with Byzantine processes. Technical Report 97/30, EPFL, 1997.

[29] P. Dutta, R. Guerraoui, and M. Vukolic. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2005.

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

[31] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, Dec. 1995.

[32] 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, Apr. 1985.

[33] 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.

[34] 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, Oct. 2002.

[35] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, jan 2005.

[36] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 289–298, July 2000.

[37] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. Krakowiak and S. Shrivastava, editors, *Advances in Distributed Systems*, number 1752 in Lecture Notes in Computer Science, pages 33–47. Springer-Verlag, 2000.

[38] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453, 2004.

[39] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, Jan. 2001.

[40] 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.

[41] I. Keidar. Challenges in evaluating distributed algorithms. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, June 2002.

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

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

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

[45] L. Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, May 1998.

[46] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

[47] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[48] B. Lampson. The abcd's of paxos. In *Proceedings of the 20th annual ACM Symposium on Principles of Distributed Computing*, page 13, 2001.

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

[50] N. Lynch. A hundred impossiblility proofs for distributed computing. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, Aug 1989.

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

[52] D. Malkhi, M. Merrit, M. Reiter, and G. Taubenfeld. Objects shared by Byzantine processes. *Distributed Computing*, 16(1):37–48, Feb. 2003.

[53] 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.

[54] J. P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 402–411, June 2005.

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

[56] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 351–360, June 2003.

[57] A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922–954, 2003.

[58] A. Mostefaoui, S. Rajsbaum, M. Raynal, and M. Roy. Condition based consensus solvability: A hierarchy of conditions and efficient protocols. *Distributed Computing*, 17:1–20, 2004.

[59] A. Mostefaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. E. Abbadi. From static distributed systems to dynamic systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 109–118, Oct. 2005.

[60] A. Mostefaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, (73):207–212, 2000.

[61] N. F. Neves, M. Correia, and P. Veríssimo. Wormhole-aware Byzantine protocols. In *2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability - Obstacles and Solutions*, June 2004.

[62] N. F. Neves, M. Correia, and P. Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[63] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[64] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 44–61, Oct. 2002.

[65] D. Powell. Fault assumptions and assumption coverage. In *Proceedings of the 22nd IEEE International Symposium of Fault-Tolerant Computing*, July 1992.

[66] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, Nov. 1983.

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

[68] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

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

[70] R. Turpin and B. A. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, Feb. 1984.

[71] 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.

[72] P. Veríssimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[73] 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.

[74] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.