Chapter 1
Stumbling over consensus research: Misunderstandings and issues

Marcos K. Aguilera

Abstract The consensus problem has recently emerged as a major interest in systems conferences, yet the systems community tends to ignore most of the large body of theory on this subject. In this chapter, I examine why this might be so. I point out misunderstandings by the systems community of the theory. I also consider some issues in this work that remains to be addressed by the theory community.

1.1 Introduction

In the consensus problem, each process proposes some initial value, and processes that do not fail must reach an irrevocable decision on exactly one of the proposed values. The consensus problem captures an essential component of replication in distributed systems: the fact that replicas (processes) need to agree on the next request they handle, so that they can remain in identical states.

The consensus problem has been a fertile topic for theoretical study and it has recently become a major interest in systems conferences. Yet, theory and practice are divorced: the large body of theoretical work on this subject has had limited impact, and the systems community tends to ignore most of that theory. In this chapter, I examine why this may be so.

The chapter is divided into two main parts. In Section 1.2, I consider some misunderstandings by the systems community of the theoretical work on the consensus problem. In Section 1.3, I consider some issues in this body of work that remains to be addressed by the theory community. Section 1.4 concludes the chapter.

The chapter presents a somewhat personal point of view. Another perspective on consensus misunderstandings is provided in [8], while [3] describes an experience of applying consensus in practice.

1.2 Misunderstandings

There are some deep misunderstandings by the systems community of a significant part of the theoretical research on the consensus problem. This section covers these misunderstandings. They have hindered the adoption of many interesting techniques, ideas, and algorithms, for incorrect reasons. My hope is that, once the misunderstandings are clarified, systems researchers can make better informed choices and benefit from work that they once thought to be inapplicable. At the same time, I hope that theory researchers can become sensitized to the misunderstandings so that they can present their research in a more effective manner.

1.2.1 Asynchronous systems

An *asynchronous (distributed) system* is a system in which processes need not satisfy any timeliness properties. There are no bounds on the relative rate of execution of processes, so one process may execute at a much faster rate than another. Moreover, there are no bounds on message delays, so messages sent from one process to another may be delivered quickly or slowly. On the other hand, in a *synchronous system*, there *are* bounds on the rate of execution of non-faulty processes and on message delays.

Critics say that asynchronous systems are not realistic, because in reality one process cannot be 10^{999999} slower than another process, and a message never takes 10^{999999} seconds to be delivered. That is a fair criticism and, indeed, it is unlikely that asynchronous systems accurately model any real system. However, asynchronous systems have an important practical aspect: algorithms developed for them are very general, because they work irrespective of whether the system is fast or slow. In contrast, algorithms developed for *synchronous* systems explicitly rely on particular timing assumptions, and the algorithms can fail if those assumptions are violated. The problem is that it is hard for the system designer to decide what timing assumptions he should make, because the timing behavior of a real system tends to be imprecise and highly variable in practice. Specifically, the average message delay of a network could be 1 millisecond, but infrequently messages may take 1 second or much longer when there is congestion. In that case, what should the system designer assume as the maximum message delay? On one hand, if he chooses 1 millisecond then this choice will be incorrect sometimes, which can cause a premature timeout and lead to consistency problems (e.g., a premature timeout may cause a backup process to be promoted to the primary, while another primary is still active). On the other hand, if the system designer picks 1 second or more as the maximum message delay, then when a message is really missing (because, say, a server or a process crashes), it will take long to timeout on the message, causing the system to block in the meantime, leading to a loss of availability. Thus, the system designer is left with two bad choices: assuming a small maximum message delay affects consistency, while assuming a large conservative delay affects avail-

ability. With asynchronous systems, the system developer does not have to choose what timing assumptions to make: he simply develops an algorithm that works irrespective of whether the system is fast or slow. From this point of view, it is much harder to criticize asynchronous systems: they merely embody the fact that timing assumptions should be avoided. The fewer the assumptions needed by an algorithm, the smaller the likelihood that it will fail when used in practice. Thus, from a practical perspective, asynchronous systems can be highly desirable when it comes to designing algorithms.

However, asynchronous systems have some practical shortcomings. Impossibility results, which state that a problem cannot be solved in asynchronous systems, are particularly limited. These results rely on the fact that asynchronous systems admit executions where messages and processes are delayed arbitrarily, whereas these executions may be unlikely. Thus, these results are of limited interest in practice; even in theory, these results are weak because they do not carry over to a system with any form of synchrony. I will elaborate on this topic, focusing specifically on the consensus impossibility result, in Section 1.2.4.

1.2.2 Eventually-forever assumptions

In the consensus literature, it is common to find assumptions in the form of eventually-forever properties. An *eventually-forever* property is a property of the form "eventually X is true and continues to be true forever after". Common examples include the following:

- *Eventual leader election.* Eventually some correct process is elected as leader and it remains leader forever after [5, 4].
- *Eventual timeliness.* Eventually non-faulty processes are timely and messages are delivered and processed in a timely fashion, and this timeliness continues forever after [6].

These assumptions are made as a condition for the algorithms to solve consensus. Practitioners object that these assumptions are not realistic and therefore the algorithms that depend on them are not useful. However, it turns out that these assumptions are actually reasonable from a pragmatic perspective. Practitioners are right that these properties cannot hold in practice, but the misunderstanding is that they are not really *required* to hold; they are only *assumed* to hold for purely technical reasons. In reality, what is required to hold are somewhat weaker properties, such as "a process remains the leader for sufficiently long". These weaker properties, however, are cumbersome to formalize, and that is why eventually-forever properties are used instead.

To illustrate this point, consider the simple example of a washing machine. Its manufacturer would like to say that, after the machine is started, it eventually terminates the washing cycle. But it will not terminate if the machine is disconnected from the power supply during operation. Hence, to ensure termination one needs an

assumption such as "eventually the machine is connected to the power supply and remains connected for 60 minutes". However, if the washing machine has a variable washing time that depends on its load, 60 minutes may not be enough and, in fact, it may be impossible to determine how long is enough without knowing the exact load. An eventually-forever property comes handy in this case: the manufacturer simply assumes that "eventually the machine is connected to the power supply and it remains connected forever after". This assumption handles the case of every possible load. Note that, once the machine terminates, it is irrelevant whether or not the machine is connected to the power supply. Saying that the machine remains connected forever is just a simple way to say that the machine is connected for sufficiently long.

Similarly, consider a consensus algorithm that uses a leader election service. The algorithm designer assumes that some process eventually gets elected as leader and remains leader forever after. The algorithm does not really need the leader for eternity, but it can be hard or impossible to know in advance for how long the leader is needed, as this can depend on many factors, such as actual message delays and the load on processes.

The washing machine manufacturer could give a table that shows, for each load, how long the machine needs to be plugged in to terminate. Similarly, algorithm designers could state assumptions that depend on all factors that influence the behavior of their algorithm. Doing so, however, requires a more refined model than the asynchronous model—something that algorithm designers prefer to avoid to keep the model simple.

1.2.3 Eventual guarantees

Many algorithms for consensus satisfy a progress guarantee described by *eventual* properties. An eventual property is a property of the form "eventually X holds". A common example is the termination property, which says that "eventually non-failed processes reach a decision". Such a property does not say exactly *when* processes reach a decision, only that sooner or later they do so.

Practitioners object that such a guarantee is not sufficient in practice, because it allows processes to terminate, say, only after 10^{99999} years. This is a valid objection, but there is a reason to do things in this way: to separate correctness from performance. As an analogy, the specification of the sorting problem requires that the algorithm eventually terminate. The exact running time of the algorithm, perhaps $O(n \log n)$, is a performance characteristic of the algorithm not a correctness guarantee, and it is good form to separate performance from correctness.

One way to address this objection is to include an analysis of the running time of the proposed algorithm, rather than just a termination proof. In an asynchronous system, this analysis can be done in terms of the maximum *observed* message delay (e.g., as in [1]), or in terms of the the largest causal chain of messages (e.g., as in [13]), or based on the time when the system starts "behaving well" (e.g., as in [6],

using a partially synchronous system with a global stabilization time). This type of analysis should be done more often.

1.2.4 The consensus impossibility result

The consensus impossibility result by Fischer, Lynch, and Paterson [7] is one of the most cited results in the consensus literature. It states that there does not exist a (deterministic) algorithm for the consensus problem in an asynchronous system subject to failures, even if messages can never be lost, at most one process may fail, and it can only fail by crashing (stopping executing).

This result is misunderstood because the exact nature of asynchronous systems is itself misunderstood. To get a better appreciation of what the result means exactly, let us examine its proof in some detail.

The proof considers a purported algorithm $\mathscr{A}$ that satisfies the safety properties of consensus[1], namely, that processes never decide differently and they never decide a value that is not proposed. It then shows that $\mathscr{A}$ violates the liveness property of consensus by constructing an execution of $\mathscr{A}$ in which processes never decide. The proof proceeds as follows. Consider the set of all possible global states of the system running algorithm $\mathscr{A}$. A state is said to be *bivalent* if the consensus decision has not been fixed yet: from that state, there are ways for processes to decide one value or another value. Note that I distinguish between the decision being known and it being fixed. For example, consider an initial state where all processes have proposed the same value v, but they have not yet communicated with each other. Then, the only possible decision is v, so the decision is certain to be v. However, the processes in the system have not yet learned that this is the case.

The proof is based on two key propositions. The first key proposition is that algorithm $\mathscr{A}$ has some initial state that is bivalent. This proposition is depicted in Figure 1.1, where the grey area represents the set of bivalent states and the leftmost disc represents a bivalent initial state. For example, in the $\diamondsuit S$-based algorithm of Chandra-Toueg [5] or in the Paxos algorithm [11],[2] one bivalent initial state is the initial state in which half of the processes proposes some value and the other half proposes a different value. In fact, this initial state is bivalent in many of the known algorithms for consensus in partial synchrony models or in models with failure detectors.

---

[1] Roughly speaking, a safety property is a property that states that something bad does not happen, while a liveness property is a property that states that something good eventually happens. The safety properties of consensus are Agreement and Validity. Agreement says that no processes decide different values, and Validity says that a process can only decide on a value that is the initial value of some process. The liveness property of consensus is Termination. Termination says that eventually correct processes decide on a value.

[2] Technical remark: to illustrate the FLP proof, here I consider the behavior of these algorithms in an asynchronous model, where the failure detector or the leader election service output unreliable information.

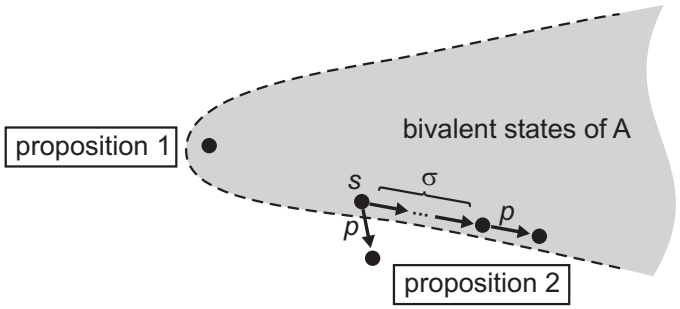

Fig. 1.1 Depiction of key propositions in the proof of impossibility of consensus.

The second key proposition is that if s is a bivalent state but after a step of some process p the state is no longer bivalent, there there exists a sequence σ of steps such that if σ is inserted before the step of process p then p's step leads to a state that is still bivalent. This proposition is depicted in Figure 1.1, where the leftmost arrow labeled p is a step of process p after state s, which leads to a state that is not bivalent; the very same step by process p, if taken after the sequence σ, leads to a state that is still bivalent. For example, in the $\diamondsuit S$-based algorithm of Chandra and Toueg, a step that leaves the grey region occurs when the last process in a majority receives the new estimate proposed by the coordinator. However, if this receipt step is delayed until after the process sees that the coordinator is suspected and abandons the round, then this receipt no longer leaves the grey region. As another example, in the Paxos algorithm, a step that leaves the grey region occurs when the last process in a majority receives a high-numbered proposal. But if this step is delayed for long enough—until another higher-number proposal appears—then the proposal becomes stale and useless.

The first and second key propositions can be shown by contradiction with relatively simple arguments, whose details are not relevant here (they are given in [7]).

These two propositions allow us to find an execution of algorithm $\mathscr{A}$ in which processes never decide. Intuitively, in Figure 1.1, the execution starts in an initial state in the grey region and all processes keep taking steps, say in a round-robin fashion. If any step by some process leaves the grey region, then one inserts a sequence of steps by other processes such that, after those steps, the aforementioned step no longer leaves the grey region. This gives us an execution in which all processes keep taking steps but the state always remain bivalent. As a result, processes never decide.

So where is the misunderstanding of the impossibility result? Most computer scientists understand impossibility results from the halting problem, which suggests that one should not even try to solve it. On the other hand, the consensus impossibility says that, given any purported solution, the consensus decision may keep getting delayed forever if processes are scheduled in an unfavorable way. This is different from the halting problem impossibility in two ways. First, the consensus impossi-

bility result is based on a model where processes can be scheduled according to the worst case, but in reality process scheduling tends to have a random aspect to it, and the probability of an unfavorable schedule could be small. Second, these unfavorable schedules produce a problem that is transient, not permanent: if processes fail to terminate because the schedule has been unfavorable, processes are still able to terminate subsequently if the schedule stops being unfavorable. A more enlightening formulation of the consensus impossibility result might be that any algorithm that ensures the safety properties of consensus could be delayed indefinitely during periods with no synchrony (the schedule is unfavorable). In fact, it can be shown that consensus is solvable when there is a very small amount of temporary synchrony in the system, namely, if there is a single link from one process to another such that this link is timely [2].

1.2.5 Uses of replication

Most people realize that a consensus algorithm lies at the heart of a service replicated using the state machine approach [10, 14]. Fewer people realize that the replicated service need not be the entire system; it could be just a smaller component of the system. For example, each node in the system may need to know some set of system parameters, such as buffer sizes, resource limits, and/or a list that indicates what machines are responsible for what function. One could use a state machine to replicate just this information across nodes. This is illustrated in Figure 1.2.

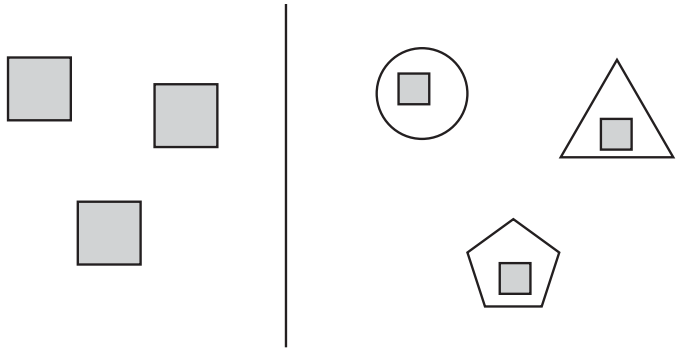

Fig. 1.2 Squares show what is replicated by consensus. On the left, the entire system is replicated. On the right, just some service within a larger (non-replicated) system is replicated.

There are some caveats in using consensus and state machines in that way. First, the number of replicas can be much larger than the minimum needed. (The minimum number of replicas needed is usually $2f+1$ or $3f+1$, depending on the failure model, where f is the maximum number of failures to be tolerated.) For example,

if one replicates the set of system parameters as described above, then the number of replicas is the number of nodes in the system, which can be very large. In this case, it is important to use simple optimizations in which not all replicas actively participate in the consensus protocol, instead of, say, using all available replicas to increase the fault tolerance threshold f. Increasing f beyond the necessary is bad because consensus protocols scale poorly in f.

The second caveat is that only requests processed through the state machine are guaranteed to see the current state of the state machine, because some replicas may be missing some state machine updates. In the above example, suppose that a component of the system wants to know the current system parameters. If it tries to read the system parameters directly from the local replica, it may obtain stale information, because the local replica may be lagging behind. To ensure it obtains up-to-date information, the component must submit a read request to the state machine and wait for the request to execute through the consensus protocol. Note that, even if the read request executes through consensus, the returned information is guaranteed to be up-to-date only for a brief moment. By the time the component uses this information (perhaps immediately after obtaining it), the system parameters may have changed already. If this is a problem then the component needs to be placed as part of the state machine, so that the component's actions can be ordered using consensus. The bottom line is that, one needs to be very careful about how components outside the state machine interact with components inside the state machine.

1.2.6 Correlated failures

The consensus problem has solutions in synchronous models, in models of partial synchrony, or in models with unreliable failure detectors. These solutions typically require that there exist an upper bound t on the number of failures in the system. Practitioners argue that these upper bounds are not realistic, even for relatively large values of t, because in practice failures could be correlated. For example, power failures, security exploits, and bugs could all result in the simultaneous failure of all processes in the system, which exceed the threshold t.

I argue that, even though correlated failures exist, there are also many situations where failures are certainly not correlated, where consensus can be useful. For instance, one could argue that tolerating crash failures is not always about tolerating crashes, but about tolerating slowness caused by busy processes, swapping to disk, or other unexpected local events. Slowness is less likely to be correlated across machines. Moreover, bugs are one of the leading cause of process crashes, and heisenbugs (bugs that are not deterministic) are probably the hardest ones to detect, and hence they are the bugs most likely to be left in a working system. Heisenbugs tend to produce failures that are not correlated.

One could argue that the techniques for handling correlated failures would automatically take care of uncorrelated failures, thus obviating the need for consensus. However, the costs of using those techniques are very different. For example, if there

is a power failure, recovery may involve rebooting the machine and retrieving state from stable storage, which can take a long time, leading to a loss of availability. In contrast, one can use consensus-based replication to handle a single (uncorrelated) node crash without any downtime.

The jury is still out on whether most failures are correlated or not. But even if many failures are correlated, I believe there is still significant benefit in tolerating those cases when they are not.

1.3 Issues

Besides misunderstandings, there are also some issues in the consensus literature that have prevented a wider adoption of existing results, algorithms, and techniques. This section covers these issues. I do not adopt an absolute notion of what is an issue—this would amount to subscribing to moral dualism. Instead, my notion of an issue is *relative* to the point of view of a practitioner who would like to benefit from the research on consensus. It is worth noting that some of the issues that I describe, particularly in Sections 1.3.3–1.3.6, extend beyond just the consensus problem: they apply to research in theory of distributed computing in general.

1.3.1 The application interface

The notion of an *interface* to an abstraction is well-known to computer scientists. For example, the problem of sorting a list has a very simple, intuitive, and agreed-upon interface. If one needs to implement the interface, it is clear what must be done, and if one wants to use the interface, it is clear how to do that. Unfortunately, such is not the case for consensus, for the following reasons:

- *Multiple application interfaces.* In order for an abstraction to be well specified, it should have a single application interface that everyone adopts. Unfortunately, consensus has two commonly adopted interfaces. The first is the interface used by the Paxos algorithm, which I shall call the p-interface. The second is the interface used by all other consensus algorithms, including algorithms based on failure detection, randomization, or partial synchrony. I shall call the latter the r-interface. There are a number of differences between these interfaces, and practitioners do not understand why there are these differences and which interface they should use. The differences are the following:

  1. *Process roles.* In the p-interface, processes are divided into proposers, learners, and deciders[3] while in the r-interface there are just processes.

[3] In the original Paxos paper, this division did not exist, but it appeared in a later description [12].

2. *Termination condition.* With the r-interface, all correct processes are required to terminate, while with the p-interface, correct processes are required to terminate only if certain conditions are met (e.g., eventually a leader is elected for sufficiently long). With the r-interface, these conditions are assumptions made in the model.
3. *Initial state of processes.* With the r-interface, *all* non-faulty processes initially propose a value, while with the p-interface, any positive number of non-faulty processes initially propose a value.

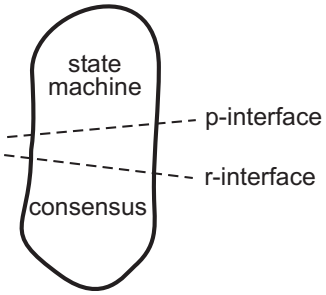

Fig. 1.3 The two different interfaces for the consensus problem.

Of these differences, the first and second are cosmetic, but the third is more significant, so let us examine it more closely. The p-interface is more directly applicable to implementing a state machine, because only one replica may receive a request for the state machine to execute, and so only one replica may propose a value. However, the r-interface can also be used to implement a state machine: when a replica receives a request v for the state machine, it sends this request to other replicas telling them to propose v if they have not proposed yet. In this way, all correct replicas will propose, as required by the r-interface. An inspection of the algorithms that implement the p-interface reveals that the first thing that a process does after proposing is to send a message to all other processes. Thus, intuitively, one can think that algorithms for the r-interface "leave out" this send-to-all step for the application to do before invoking consensus. This difference essentially corresponds to two different cuts in the boundary between a state machine and consensus, shown in Figure 1.3. The r-interface cuts at a lower level, requiring that the state machine perform a send-to-all before invoking consensus. With the p-interface, this send-to-all is effectively done by the consensus algorithm.

So this difference is not inherent. However, it is problematic because it makes it difficult for practitioners to understand the consensus literature. For example, there is an algorithm for the r-interface in which all non-faulty processes decide in one communication step if all processes propose the same value. At the same time, one can show that this is not possible for the p-interface. (Intuitively, this is

because with the r-interface, all correct process can initially send their proposed value to all and then wait to receive $n-f$ messages, but this initial waiting is not possible with the p-interface since only a few processes may propose.) These two results look contradictory, but they are not.

The reason for having two interfaces is historic. The r-interface appeared as a variation of the interactive-consistency problem, in which every process starts with some initial value. The p-interface was later proposed as an interface more directly applicable to the state machine approach. Regardless of the historical development, it is about time to converge on a single interface for consensus.

- *Usability issues.* Another problem with the consensus interface is usability, and this problem has multiple facets.

  First, consensus is a single-use service: after decision is reached, the consensus instance is stuck and must be eventually garbage collected. Garbage collection must be done manually by the application, and this can be tricky, because a process that crashes before the decision may subsequently recover and want to learn the decision, but it cannot do that if the consensus instance has been garbage collected. Another problem with a single-use service is that it needs to be instantiated repeatedly, which imposes overhead and impacts performance. This can be an issue for applications that must solve consensus repeatedly at a high rate.

  Second, the consensus problem assumes that the set of processes that propose and decide (the participants) is fixed and known a priori. However, in practice, a machine may crash and remain crashed for long periods (even permanently), and this machine must be excluded from the set of participants and eventually be replaced. How to do that is explained in an extension of Paxos to multiple instances, called Multi-Paxos, but Multi-Paxos is an algorithm not an interface. This feature needs to be described by the interface, by specifying it independently of how it is implemented. Furthermore, Multi-Paxos has several important optimizations but these optimizations are not expressible through the consensus interface: they apply neither for a (one-instance) consensus algorithm nor for multiple instances of a consensus algorithm given as a black box.

The above problems perhaps indicate that consensus is the wrong abstraction to expose. In other words, the consensus problem does not have the same simple and universal appeal that the sorting problem has. Consensus may not be the most intuitive and applicable abstraction for practitioners. An alternative to consensus is the atomic multicast abstraction as defined in [9], which provides reliable and totally-ordered delivery to a variable subset of users which need not include the broadcaster. In my opinion, this abstraction should be studied and adopted more often in the theoretical literature.

1.3.2 Violation of abstraction boundaries

The consensus literature often states key properties and results by referring to the inside of a consensus algorithm, instead of referring to the interfaces exposed by the

consensus algorithm. This is problematic for practitioners because often they do not want to know what is "inside the box"; they only care about how the box behaves externally.

A common example is to analyze the performance of a consensus algorithm in terms of the number of phases that it needs to decide, where a phase is an algorithm-specific notion. This metric looks inside the box and it is not useful to compare different algorithms, since each algorithm may have its own notion of what is a phase, or may not have phases at all.

Another common example is to state the timeliness requirements of an algorithm in terms of a communication primitive implemented by the algorithm itself. For example, some consensus algorithms in a partially synchronous system require that some non-faulty process be able to receive responses to its queries in a timely fashion. However, a query is an algorithm-specific notion.

To follow the principles of abstraction, results and properties about an algorithm should always be stated in terms of the algorithm's upper and lower interfaces. The upper interface is the consensus application interface, while the lower interface refers to the services on top of which the consensus algorithm runs, such as interprocess communication and leader election or failure detection. Rather than analyzing the performance of an algorithm in number of phases, it is more useful to do so in terms of the lower interface—say, based on the maximum observed message delay, in executions where the failure detector makes no mistakes (e.g., as in [1]). Rather than stating the synchrony requirements of an algorithm in terms of timeliness of a query-response mechanism implemented by the algorithm, it is more useful to do so in terms of properties in the lower interface—say, by indicating which links in the message-passing service are timely (e.g., as in [2]). If this is not possible, then the query-response mechanism needs to be placed in the lower interface so that it is exposed *outside* the algorithm.

1.3.3 Ambiguities and errors

In distributed computing, it is easy to make mistakes in algorithms and results. This is because distributed systems are inherently non-sequential and failures create exceptional conditions, resulting in many corner cases that are easy to overlook. This, in turn, leads to technical glitches, ambiguous results, or even more serious mistakes in published results.[4] These problems are eventually detected (and perhaps corrected), but many times they are only publicized informally through word of mouth. This method may suffice for the researchers in the area, but may not reach practitioners and outsiders, making it very hard for them to understand the literature.

One proposal to address this issue is to publish the mistakes and possible corrections in the form of small notes. These notes could appear as short papers or brief

[4] Lest this discussion be interpreted as a remark about any particular paper, I note that there are many results about which this concern can be raised.

announcements in one of the important theory conferences in distributed computing. This would create a public record of the problem for outsiders (and insiders) to be aware of.

1.3.4 Unfriendly formalisms

As a reaction to ambiguous algorithms and incorrect results, the theoretical community has proposed the use of formal frameworks to present algorithms and results, and to prove their correctness. These formalisms certainly eliminate ambiguity and reduces errors, but they are difficult for practitioners to digest because they are too low-level or abstract.

A common practice is to explain results both intuitively and formally, in order to reap the benefits of both approaches. However, it is dangerously easy to provide intuitions that are much too superficial, using as justification a formal presentation "given later". It is also dangerous to provide formal presentations that are much too low-level, using as justification the intuition "given earlier". The result is that neither intuitive nor formal presentations end up being useful.

Researchers in the theoretical community must find the right balance between formalism and intuition for the particular result that they want to convey. A one-size-fits-all solution cannot adequately address every case, or even most cases.

1.3.5 Lack of feedback from practitioners

Practitioners provide very little feedback to theoretical algorithm designers on what needs to be improved, and algorithm designers rarely seek feedback from practitioners. As a result, one often finds algorithm designers optimizing for many different variations of cases and parameters, without knowing which are relevant. That is a somewhat inefficient way to proceed, because only a few cases and parameters require optimization in practice. It would be much better to get input on an actual system from practitioners on what is not working well, and thereby focus on improvements that are likely to be used. Perhaps part of the problem is that it is often difficult to understand a system and isolate which aspects are likely to benefit from better algorithms. Doing so requires close collaboration between theoreticians and practitioners.

1.3.6 Hidden limitations in algorithms

Sometimes algorithms proposed in the literature have hidden limitations that could be troublesome in practice. For example, there are many consensus algorithms that

decide quickly if some fixed designated process is the leader ("process 1"), but these algorithms become much slower if this process crashes or is not the leader. Such limitations can often be circumvented, but only if practitioners are aware of them. For that to happen, algorithm designers need to be explicit about the weaknesses of their algorithms.

In some cases, the limitations of an algorithm are inherent and cannot be overcome, and whether they are tolerable in practice depends on factors that only practitioners can determine. In these cases, it is even more important for them to be disclosed. If limitations are hidden, a practitioner that implements the algorithm eventually finds out the limitation by herself, but only after much effort. At that point, the practitioner will conclude that either the algorithm designer could not see the problem or, perhaps worse, he was trying to hide it.

Limitations of an algorithm should be explained by the designer of the algorithm, when the algorithm is published. It is better if the designer tell readers of a limitation than if readers later tell the designer.

1.4 Conclusion

The consensus problem is at the heart of replicated distributed systems, which are increasingly becoming a vital part of our society in areas such as commerce, banking, finance, communication, critical infrastructure, and others. While consensus has recently attracted the attention of the systems community and practitioners, the theoretical work on this problem remains underutilized due to misunderstandings and issues. This situation is a loss for everyone: theoreticians are missing an opportunity to apply their work, while practitioners are overlooking an untapped resource. To address this problem, the theory and practical communities need to engage in a more open dialog. This step is sorely needed. The conceptual mechanisms and techniques underlying the consensus problem are very subtle and, without a firm theoretical foundation, it will be hard to go very far. At the same time, consensus is a problem initially motivated by practical concerns, and without interest and feedback from practitioners, the theory will have limited impact.

Acknowledgements I am grateful to Ken Birman and the anonymous reviewers, who provided many valuable comments that helped improve this chapter.

References

1. Marcos K. Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, April 2000.
2. Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *ACM Symposium on Principles of Distributed Computing*, pages 328–337, July 2004.

3. Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live—an engineering perspective. In *ACM symposium on Principles of distributed computing*, pages 398–407, August 2007.
4. Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
5. Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
6. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
7. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
8. Rachid Guerraoui and André Schiper. Consensus: the big misunderstanding. In *IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 183–188, October 1997.
9. Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornelpublisl University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
11. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
12. Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.
13. André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.
14. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.