Consensus: the Big Misunderstanding*

Rachid Guerraoui André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
e-mail: {guerraoui,schiper}@di.epfl.ch

Abstract

The paper aims at clarifying some misunderstandings about the consensus problem. These misunderstandings prevent consensus from being considered as it should be, i.e., a fundamental paradigm in the context of fault-tolerant distributed systems, not only from a theoretical point of view, but also from a practical point of view. Six frequent misunderstandings are discussed.

Misunderstanding 1: Consensus is for theoreticians only

Consensus can be viewed as a general form of agreement in distributed systems [17]. The problem is defined over a set of processes $\{p_1, p_2, \ldots, p_n\}$: each process p_i has an initial value v_i, and the correct processes (those that do not crash) have to decide on a common value v that is the initial value of one of the processes [3]. This problem has attracted theoreticians for over 15 years and has resulted in a large body of work, the most known being the Fischer, Lynch and Paterson result proving that consensus is not solvable in an asynchronous system[1] if a single process may crash (the so called FLP impossibility result) [9]. Apart from this result, many solutions to the consensus problem have been described in other system models (synchronous [6], partially synchronous [7], asynchronous with failure detectors [3], etc).

While the consensus problem has attracted much attention in the theoretical distributed systems community, it has been largely ignored by systems' implementors. Implementors usually consider the consensus problem to be irrelevant for real systems. It is frequently argued that:

- *Real systems have to solve practical agreement problems such as atomic broadcast (also called total order broadcast), atomic commitment, leader election, group membership, etc. So why worry about the consensus problem and the FLP impossibility result?*

Such a claim, stating that results applying to the consensus problem are irrelevant to other agreement problems, is incorrect. The simplest example is atomic broadcast: the atomic broadcast problem and the consensus problem have been shown to be equivalent [3]. Equivalence means that (i) any solution to the atomic broadcast problem can be used to solve the consensus problem[2], and (ii) any solution to the consensus problem can be used to solve the atomic broadcast problem[3]. Because of (i), the atomic broadcast problem is also subject to the FLP impossibility result. Because of (ii), a solution for the consensus problem can be used as a building block to solve the atomic broadcast problem. This last statement is usually turned down with the following argument: *it is not because consensus* can *be used to implement atomic broadcast, that consensus has to be used to implement atomic broadcast.* However, because of (i), the inherent difficulty of solving the consensus problem inevitably ap-

[1] An asynchronous systems does not bound the transmission delay of messages, nor the relative process speeds.

[2] This is easy to show: (1) initially, every process p_i atomically broadcasts its initial value v_i; (2) the decision is the first value v delivered.

[3] (i) is called *reduction* of the atomic broadcast problem to the consensus problem, and (ii) is called *reduction* of the consensus problem to the atomic commitment problem.

plies to the atomic broadcast problem, independently of the solution that is used.

While atomic broadcast is equivalent to consensus, the same result does not hold for all agreement problems. However, this does not reduce the difficulty of the other agreement problems, which are at least as hard to solve as consensus (e.g., atomic commitment, see [10]). A rigorous approach to consensus, which can be used to address other agreement problems [11], is thus of primary importance when building fault-tolerant distributed systems.

Misunderstanding 2: Time-outs are enough

The FLP impossibility result applies to asynchronous systems, in which there is no time. For this reason, the FLP impossibility result is often considered as an obvious result: without time, there is no way to detect crashed process. So it is frequently argued that:

- *Consensus can easily be solved by adding time-outs to the asynchronous system model.*

This statement is flawed. The absence of time in the asynchronous system model does not prevent processes from suspecting the crash of other processes. A time-out mechanism can easily be implemented based on a (purely local) logical time: the logical time of process p_i can be defined as the number of instructions that p_i has executed. In other words, adding time-outs to the asynchronous system model is not enough to overcome the FLP impossibility result [6]. The heart of the FLP result is the impossibility to distinguish crashed processes from those that are slow or connected via slow links.

Understanding that time-outs are not enough to overcome the FLP impossibility result, does not lead implementors to care about it. The usual argument is the following:

- *The FLP impossibility result has been proven in an unrealistic system model: the scenarios used to prove the impossibility result never occur in a real system.*

This is probably the most interesting point to discuss. While the FLP impossibility result holds in an asynchronous system, most implementations assume "implicitly" a stronger system model. Typically, a LAN with a time-out of 30 seconds to detect crashed processes might adequately be modelled as a synchronous system (with infrequent timing failures). In this case the FLP impossibility result does not hold[4].

Nevertheless, there is an inevitable trade-off between (1) reducing the probability of timing failures (i.e., the probability of incorrect failure suspicions), and (2) fast reaction to process crashes. This trade-off is at the heart of the misunderstanding. Consider atomic broadcast implemented using a sequencer process (e.g., Amoeba [12], Isis [1]), and a time-out of 30 seconds to suspect the crash of the sequencer process. In this case, at least 30 seconds are needed to react to the crash of the sequencer process, i.e., *the crash of the sequencer will lead to a black-out period of at least 30 seconds.* This might be unacceptable for time-critical applications. On the other hand, reducing the time-out value increases the probability of incorrect failure suspicions! The probability of false suspicions might still be low with a time-out of 15 seconds, but it can be high with a time-out of a 1/2 second! As soon as the probability of incorrect failure suspicions becomes non-negligible, it is no longer adequate to consider the system as being synchronous. In this case the FLP impossibility result becomes relevant, and a rigorous solution to the consensus problem (and to consensus related problems) that tolerates frequent incorrect failure suspicions, becomes mandatory.

To summarise, (1) the difficulty of solving consensus and (2) a fast reaction to process crashes – in order to reduce the black-out period of the system – are closely related.

Misunderstanding 3: There is no life after FLP

Having understood the fundamental difficulty of solving consensus, one could make the following observation:

- *Consensus is sometimes solvable, and sometimes not: so why care about the latter case? Just live with it!*

Indeed, the FLP impossibility result states that in an asynchronous system there is no algorithm that solves consensus *in every possible run*[5]. Given any algorithm $\mathcal{A}$, the algorithm can at best solve consensus in a subset of all possible runs, which we will denote by $\mathcal{A}$-*runs*[6]. A different algorithm $\mathcal{A}'$ might solve consensus in a different subset of runs, denoted by $\mathcal{A}'$-*runs*. If

[4]However, in this case it is not appropriate to characterize the system as being asynchronous!

[5]We ignore here the probabilistic algorithms.

[6]$\mathcal{A}$-*runs* are usually called the *admissible* runs of $\mathcal{A}$.

$\mathcal{A}$-*runs* $\subset$ $\mathcal{A}'$-*runs*, then algorithm $\mathcal{A}'$ can be seen as more resilient than algorithm $\mathcal{A}$.

In other words, *the FLP impossibility result forces us to characterize the set of runs in which a given algorithm solves consensus.* This is well understood in the synchronous system model, in which an algorithm $\mathcal{A}$ for solving an agreement problem is proven correct only in runs with no more than f failures (process crashes, channel omission failures, channel timing failures)[7].

Several researchers have been working on weakening the assumptions of the synchronous model (bounded number of failures) in the context of the consensus problem. This has led to the definition of various system models: partially synchronous [7], timed asynchronous [5], asynchronous with failure detectors [3]. These models have led to algorithms that never lead processes to disagree on the decision value (i.e., the safety property is never violated). The partially synchronous model and the timed asynchronous model, both assume a bound δ on the transmission delay of messages and do not bound the number of timing failures. However, they put the following restrictions on runs:

- the partially synchronous model assumes the existence of some time t after which there are no longer timing failures[8];
- the timed asynchronous model assumes that the systems goes through *stable* periods, and *unstable* periods. A stable period is a period during which no timing failures occur.

The asynchronous system model augmented with failure detectors does not assume any bound on the message transmission delay, but characterizes the incorrect failure suspicions that can occur during a run. Consider the failure detector $\diamond\mathcal{S}$ [3]:

- in an asynchronous system augmented with the failure detector $\diamond\mathcal{S}$, there is a time t after which one correct process is no more suspected by any correct process.

To summarise, the FLP result has led to the definition of various models (partially synchronous model, timed asynchronous model, asynchronous model with failure detectors) that are less restrictive than the synchronous model, and allow us to characterize the runs under which a given algorithm solves consensus.

[7]A major drawback of algorithms based on the synchronous system model is that, whenever a run does not satisfy the bound f, the safety property of the consensus problem is violated (i.e., two processes can disagree on the decision value). Notice that this is not the fault of the synchronous system model, but of the algorithms that have been developed based on the assumptions of the synchronous model.

[8]We discuss the "no more" assumption in the next section.

Misunderstanding 4: The failure detector model is unrealistic

Among the three models mentioned above, the asynchronous model with failure detectors is the most simple and the most general. The simplicity of the model has allowed the proof of an important minimality result [2]: $\diamond\mathcal{S}$ is the weakest failure detector that allows us to solve consensus in an asynchronous system[9]. In other words, an algorithm $\mathcal{A}$ that solves consensus in an asynchronous system with the failure detector $\diamond\mathcal{S}$ is optimal: there exists no algorithm $\mathcal{A}'$ and no run r such that $\mathcal{A}'$ solves consensus in run r, but not $\mathcal{A}$.

However, the failure detector model (in the context of asynchronous systems) is often not well understood and the following criticisms are expressed:

1. *The failure detector $\diamond\mathcal{S}$ is not implementable in an asynchronous system.*
2. *The model does not incorporate process recovery: in real systems processes do recover.*

We discuss each of these criticisms.

1. The question about the implementability of the $\diamond\mathcal{S}$ failure detector is equivalent to the following question, in the context of the timed asynchronous model (see "Misunderstanding 3"): *can stable periods be implemented?* Obviously the question does not make sense. The question about the implementability of $\diamond\mathcal{S}$ is exactly of the same nature! The definition of $\diamond\mathcal{S}$ must be seen as a specification for the implementation of the failure detection mechanism: the time-out value chosen should be as small as possible (if fast reaction to process crashes is required), but not too small, to guarantee the properties of $\diamond\mathcal{S}$ with a probability close to 1.

Providing such a guarantee might appear to be impossible, because $\diamond\mathcal{S}$ *requires the existence of a correct process p and a time t after which p is no more suspected by any correct process.* In other words, after t, the property must hold "forever". Can such a property be ensured? Actually, requiring that a property

[9]The failure detector $\diamond\mathcal{W}$ is usually presented as the weakest for solving consensus. However, $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ are equivalent [3].

holds "forever" is only necessary from a formal point of view, but not from a pragmatic point of view. From a pragmatic point of view, "forever" means "until the problem (e.g., consensus) is solved"[10].

2. The consensus problem, and the failure detectors, have been defined in a model in which processes do not recover. This simplifies the specification. Take for example a model with process recovery and a process p that crashes and later recovers. *Is p a correct process*? This is an important question, because the specification of the consensus problem requires that every "correct" process eventually decides. Thus, if the answer is "yes", p must eventually decide; if the answer is "no", p is not obliged to decide. However, the "crash/no recovery" model can be extended to include process recovery. Solving consensus in a "crash/recovery" model using failure detectors is discussed in [14]. The solution is very close to the solution in the "crash/no recovery" model.

Misunderstanding 5: Time-free means inefficient

The partially synchronous model and the timed asynchronous model are both time based models. The asynchronous model with failure detectors is a "time-free" model (time is hidden in the failure detectors). It is frequently argued that:

- *Solving consensus in a time-free model leads to a complex and inefficient solution.*

We start by discussing the complexity issue. To make the discussion concrete, we consider the Chandra-Toueg consensus algorithm based on the failure detector $\diamond\mathcal{S}$ [3]. The algorithm has multiple rounds, with a different process acting as the coordinator in every round (the so-called rotating coordinator paradigm). The algorithm indeed requires some time to be fully understood, because it is intrinsically complex. Complexity should however not be mixed up with inefficiency. The algorithm is complex because it can tolerate an unbounded number of incorrect failure suspicions. However, in a run with no process crashes and no failure suspicions (we call such runs *good runs*), the Chandra-Toueg consensus algorithm solves consensus in the first round! It is legitimate to evaluate the efficiency of a consensus algorithm in such *good runs*[11].

Defining a measure for the efficiency of a distributed algorithm is not easy. The number of messages sent by an algorithm $\mathcal{A}$ is one possible measure of the efficiency of $\mathcal{A}$. The number of communication steps (or *latency degree*, see [15]), is probably a more adequate measure. Taking into account both the number of messages and the number of communication steps is an even better measure for the efficiency of an algorithm. To keep things simple, we measure here the efficiency of a consensus algorithm as the number of communication steps in good runs.

Solving consensus with three communication steps. The Chandra-Toueg consensus algorithm based on $\diamond\mathcal{S}$ requires three communication steps in good runs (Fig. 1)[12]:

- initially, process p_1 broadcasts its initial value v_1 (p_1 proposes v_1 as the decision value);
- a process accepts p_1's proposal by sending an *ack* back to p_1;
- once p_1 has received a majority of *ack* messages, it broadcasts a *decide* message.

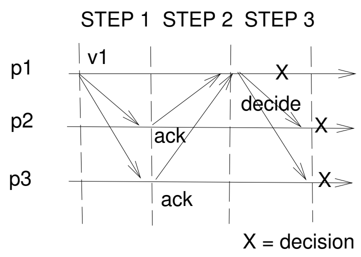

Figure 1: The Chandra-Toueg consensus algorithm based on $\diamond\mathcal{S}$ (in good runs).

Solving consensus with two communication steps. The Chandra-Toueg consensus algorithm can be improved. The early consensus algorithm [15], also based on the $\diamond\mathcal{S}$ failure detector, solves consensus in only two communication steps in good runs (Fig. 2):

- initially, process p_1 also broadcasts its initial value v_1;

[10] This is obvious: who cares what happens once a problem is solved. However, from a formal point of view, the specification "until the problem is solved" is not adequate, because it mixes the specification of failure detectors, with its use to solve a problem.

[11] If the failure detectors are implemented with adequate timeouts, and if crashes are rare, then good runs are by far the most frequent runs.

[12] Figure 1 takes into account an obvious optimisation of the first round of the Chandra-Toueg consensus algorithm.

- every process accepts p_1's proposal by forwarding v_1 to all. A process decides after having received v_1 from a majority of processes.

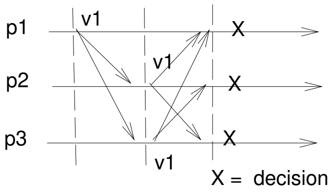

Figure 2: The early consensus algorithm based on $\diamond\mathcal{S}$, in good runs.

Solving consensus with only one communication step? Solving consensus in two communication steps in good runs is not so bad! Is it possible to do better? Better means solving consensus in a single communication step. If this is not possible, then the early consensus algorithm is optimal in the number of communication steps in good runs!

Our intuition is that there is no algorithm, whatever (realistic) system model, even time-based, that solves consensus with a single communication step in good runs[13]. Typically, such an algorithm would decide in good runs on the initial value of one of the processes, e.g., p_1 (see Figure 3). However, such an algorithm cannot be correct. A process, after having received v_1, cannot be sure that the other processes also have received v_1 (and decided on v_1). Consider the following scenario:

- process p_1 broadcasts v_1;
- simultaneously, a network failure separates the processes in two partitions, one partition including $\{p_1, p_2\}$, the other including $\{p_3, p_4, p_5\}$. Assume that only p_2, and of course p_1, receive v_1 and decide v_1. Processes p_3, p_4 and p_5 are likely to decide on a value different from v_1, i.e., the agreement property of the consensus problem is violated.

One might be tempted to fix the problem in a time-based model as follows:

- whenever some process p_i has received p_1's initial value v_1, process p_i waits for some duration Δ before deciding (where Δ is the duration needed to detect that a network failure might have occurred). After Δ, if p_i has not been notified of a network failure, then p_i decides on v_1.

[13] Notice that the algorithm given in [8], based on the timed asynchronous model, appears to have the same number of communication steps as the Chandra-Toueg algorithm based on $\diamond\mathcal{S}$.

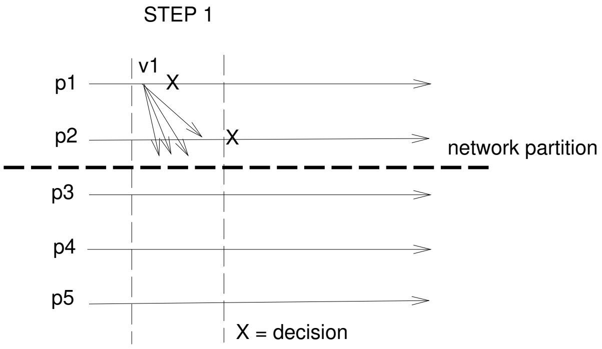

Figure 3: Algorithm with a single communication step in good runs: the agreement property of consensus is violated.

This introduces a hidden cost Δ in the consensus algorithm, which has to be added to the one step communication cost of the algorithm. A careful analysis is needed to compare the cost of (i) two communication steps vs (ii) one communication step plus Δ. However, a result showing that (ii) is less costly would be surprising, since the delay needed to detect a communication failure is at least the delay needed to transmit a message.

Misunderstanding 6: Asynchronous algorithms cannot be used for time critical applications

It has been argued in [4] that:

- *asynchronous atomic broadcast algorithms cannot be used in critical applications, because they do not guarantee a bound on the time it takes to broadcast a message in the presence of failures.*

This argument is incorrect. If an algorithm $\mathcal{A}$, based on a model M, guarantees a bound on the time it takes to solve atomic broadcast (or any agreement problem), this is because the model M restricts the number of failures (e.g., the synchronous system model), and is not due to a more sophisticated algorithm $\mathcal{A}$.

There might indeed be a difference between two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ based on two different system models, M_1, respt. M_2. If only one of them, say M_1, incorporates a model for resource allocation (process scheduling, network allocation), then the algorithm $\mathcal{A}_1$

might be more adequate for time critical applications than the algorithm $\mathcal{A}_2$ based on a different model. However, we know of no atomic broadcast algorithm (and more generally of no agreement algorithm) based on such a model.

We believe that the right approach when building agreement algorithms for time critical applications, is a two phase approach:

- *Phase 1.* Develop an algorithm $\mathcal{A}_{tf}$ in a time free model (asynchronous model with failure detectors), and prove it correct (safety, liveness);
- *Phase 2.* "Immerse" the algorithm in a real system, e.g., a system with real-time clocks. Real-time guarantees can for example be obtained as the result of the immersion of the algorithm $\mathcal{A}_{tf}$, by taking into account the specific properties of the real system.

This argument can be found in [13], where Phase 1 is called "design" phase of the algorithm and Phase 2 is called "implementation" phase. In the implementation phase, details not present in the design phase are decided, e.g., the implementation of the failure detectors (heartbeat messages vs. ping/I-am-alive messages, values for time-outs, etc). Such implementation decisions depend obviously on the characteristics of the underlying network. A timeliness analysis of the time-free algorithm $\mathcal{A}_{tf}$ can then be done, based on the characteristics of the underlying network (see for example [16]).

Conclusion

While discussing frequent misunderstandings about solving consensus and consensus related problems, the paper has advocated an approach in which agreement problems (e.g., consensus) are solved in the most general model (asynchronous with failure detectors).

Such a general approach does not lead to loss of efficiency and should be adequate even in the context of time critical applications. Timeliness guarantees include liveness guarantees: therefore proving liveness should be the first step before considering timeliness properties.

References

[1] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.

[2] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of ACM*, 43(4):685–722, 1996.

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

[4] F. Cristian. Synchronous Atomic Broadcast for Redundant Broadcast Channels. *The Journal of Real-Time Systems*, 2:195–212, 1990.

[5] F. Cristian and Ch. Fetzer. The Timed Asynchronous System Model. Technical Report CSE97-519, Department of Computer Science, UCSD, 1997.

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

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

[8] Ch. Fetzer and F. Cristian. On the Possibility of Consensus in Asynchronous Systems. Technical Report CSE95-415, Department of Computer Science, UCSD, 1995.

[9] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.

[10] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *9th Intl. Workshop on Distributed Algorithms (WDAG-9)*, pages 87–100. Springer Verlag, LNCS 972, September 1995.

[11] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *IEEE 26th Int Symp on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, June 1996.

[12] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *IEEE 11th Intl. Conf. Distributed Computing Systems*, pages 222–230, May 1991.

[13] G. Le Lann. On Real-Time and Non Real-Time Distributed Computing. In *9th Intl. Workshop on Distributed Algorithms (WDAG-9)*, pages 51–70. Springer Verlag, LNCS 972, September 1995. Invited paper.

[14] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the Crash-Recover Model. Technical report, EPFL, Dept d'Informatique, July 1997.

[15] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[16] N. Sergent. Performance Evaluation of a Consensus Algorithm with Petri Nets. In *7th Intl. Workshop on Petri Nets and Performance Models*, pages 143–152. IEEE Computer Society, June 1997.

[17] J. Turek and D. Shasha. The Many Faces of Consensus in Distributed Systems. *IEEE Computer*, 25(6):8–17, June 1992.