Generic early-deciding techniques for Consensus

Piotr Zieliński
University of Cambridge
Computer Laboratory
piotr.zielinski@cl.cam.ac.uk

March 29, 2004

(the author is a student)

Abstract: Two techniques for obtaining efficient agreement protocols are described. Firstly, Consensus frameworks, such as Lambda [7], allow one to easily construct latency-optimal agreement protocols tailored to particular settings. Unlike Lambda, our framework consists of rounds that do not have to terminate and thus are easier to implement. It can also handle Byzantine failures.

Our second technique presented exploits the Validity property of Consensus and we used it to design an atomic broadcast algorithm that is one communication step faster than any other known algorithm [13].

Keywords: Consensus frameworks, agreement protocols, Byzantine faults

Contact author: Piotr Zieliński
University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UK

Email: piotr.zielinski@cl.cam.ac.uk
Phone: +44 1223 7-63691
Fax: +44 1223 3-34678
1 Introduction

The Consensus problem amounts to reaching a common decision in a distributed environment where \( f \) out of \( n \) processes can fail. The FLP impossibility result [6] states that this problem is unsolvable in purely asynchronous settings, even if only one process can fail. As a result, a great deal of Consensus-related research has focused on minimally extending the model (e.g., with failure detectors [3], randomization [1], or time constraints [4]), and designing Consensus algorithms optimized for the worst case scenarios allowed by the model. This paper takes a different angle and investigates methods of designing algorithms that are efficient in common scenarios.

Several such early-deciding algorithms have already been proposed for both the crash-stop model [8, 11] and the Byzantine one [2]. These algorithms are model-specific in the sense that they assume a particular definition of a “common scenario”. No single algorithm can perform optimally in all “common scenarios” because, for example, oracle-efficiency and zero-degradation cannot be achieved at the same time [7]. Therefore, if the assumptions change, one has to design a new algorithm or at least re-prove the correctness of the original one. This is usually a non-trivial task, because these algorithms consist of many interdependent rounds, and rely on various eventual assumptions, such as failure detectors being eventually accurate [3].

To avoid this effort, and contribute to a better understanding of Consensus in general, we propose a sketch of a modular framework for constructing latency-optimal agreement protocols. It consists of a sequence of easily replaceable OTC modules, described in Section 2. In Section 3, we compare this approach with the similar framework Lambda [7]. Finally, Section 4 presents another protocol-independent idea for reducing latency in Consensus-based systems.

2 Optimistically Terminating Consensus

Our framework is based on a new Optimistically Terminating Consensus primitive (OTC), which – similarly to condition-based methods [10] – guarantees termination only in favourable circumstances. This property, however, is not sufficient to make different instances of OTC reach the same decision, which is necessary to solve Consensus if the circumstances are only eventually favourable. Therefore, we additionally require that if a decision has been made, then its value should always be determinable by an observer.
who has *simultaneous* access to the local states of all the correct processes. (The observer does not know which processes are correct, only that all of them are among the processes whose local states he can access.) For Byzantine settings, we assume that faulty processes can lie about their local states.

Standard Consensus can be implemented with a sequence of OTC instances. Before an instance $i$ actually begins, each process stops participating in all previous instances, so that the part of its local state corresponding to these instances does not change any more. This *frozen* state is broadcast, therefore each process will eventually learn the *current* local state of all previous instances at all correct processes. Recall that with OTC, if a decision has been made by any of the previous instances, processes will be able to determine its value, and use it as their proposals for instance $i$. On the other hand, if no decision could have been made by any previous instance, then processes are free to propose any value. This method, based on the first round of Paxos [8], ensures that decisions made by different OTC instances are the same. If the circumstances are eventually favourable, some instance $i$ will reach a decision.

In a simple implementation of OTC instances, we assume that each of them has a single coordinator. As in the second round of Paxos, the coordinator broadcasts its proposal to other processes for *acceptance*. A process *decides* if the proposal has been accepted by more than $\frac{n}{2}$ processes.

The situation is slightly more complicated in Byzantine settings, where the coordinator's proposal is first merely *weakly accepted* by other processes. The proposal is *accepted* once more than $\frac{n+f}{2}$ processes have weakly accepted it. Finally, it becomes the *decision* if it has been accepted by more than $\frac{n+f}{2}$ processes [2]. Recently, we discovered [12] that a proposal weakly accepted by more than $\frac{n+3f}{2}$ processes can also become decision. This allows us to skip the accepted stage, thereby achieving two-step latency in failure-free runs. The existence of such an algorithm was suggested in [9].

As we can see, the rules for “upgrading” a given value from *weakly accepted* to *accepted*, and then to a *decision* are surprisingly simple, even in Byzantine settings. To adapt the algorithm to particular requirements, only these rules need to be modified – the rest of the framework remains unchanged. Proving that a given set of rules is correct can be difficult and tedious so, in a future paper, we will present an alternative: a simple set of sufficient conditions against which the given set of rules can be tested. These conditions enable us to easily prove the correctness of existing algorithms [5, 7] and to design new latency-optimal protocols for both crash-stop and Byzantine models.

### 3 OTC framework vs. Lambda

All the algorithms implemented in the Lambda framework [7] can be expressed in the OTC framework without loss of efficiency. In addition, the approach advertised in Section 2 has the following advantages over Lambda.

Firstly, the implementations of OTC instances are completely independent, which makes them conceptually simpler than modules in Lambda. They can be replaced on a per-instance basis, so it is possible to use different implementations of OTC instances in the same run of Consensus. For example, one might use an implementation that is fast in good runs for the first instance, and a more fault-tolerant one for the others.

Secondly, the OTC abstraction is implementable in purely asynchronous settings. All external factors, such as choosing the proposals and the starting time for a new instance, are clearly separated from the implementation of the OTC instances. As a result, they can also be modified independently from the rest of the algorithm.
Thirdly, OTC instances do not have to terminate. This makes them easier to implement because, instead of worrying about liveness at every place in the algorithm, one can just trust the next instance to stop the current one in due time. For example, assume that an instance is supposed to decide (in one step) if \( n - f \) processes have proposed the same value [7]. An OTC instance can just wait for any \( n - f \) identical proposals. On the other hand, if the first \( n - f \) received proposals are not the same, a Lambda module must abort because it could jeopardize progress by waiting for more.

The Lambda framework is “forward-consistent”, that is, each round is responsible for setting the correct proposals for the next one. On the other hand, our framework is “backward-consistent”, so it is the responsibility of the current round (instance) to ensure consistency with all the previous ones. We believe that the latter approach is better suited for Byzantine settings, because it clearly makes the instance itself responsible for its own correctness. For this reason, we expect a possible Byzantine version of Lambda to be more complicated than our framework.

4 Using Validity to improve performance

The Validity property of Consensus states that the decision must have been proposed by one of the participants. This fact can be used to improve the performance of some Consensus-based systems, regardless of how Consensus is actually implemented.

Assume that one participant, say \( p \), can propose any value \( v \), whereas the others can propose only a special symbol \( \perp \). In this case, Validity implies that the only possible decisions are \( v \) and \( \perp \). In particular, if even \( p \) proposes \( v = \perp \), then processes can decide on \( \perp \) in one communication step. By defining \( \perp \) as “no messages to send” (the most common case), we constructed a latency-optimal atomic broadcast algorithm for closed groups [13], which is one communication step faster than other known algorithms.

The latency can sometimes be reduced even if \( v \neq \perp \), namely, if the decision (\( v \) or \( \perp \)) is processed by another algorithm, and \( p \)'s choice of \( v \) is so that the output of this algorithm is the same for both \( v \) and \( \perp \). In a future paper, we will describe how to use this observation to improve the performance of our atomic broadcast algorithm [13] in runs with failures, while preserving the same optimal latency in failure-free runs.

5 Conclusion

This paper presented two methods of designing fast and simple agreement protocols. Firstly, we introduced a new abstraction, Optimistically Terminating Consensus, and used it to construct a framework for designing Consensus algorithms optimized for a variety of different “common scenarios”. Secondly, we showed how to use the Validity property to reduce latency.

In our opinion, Consensus frameworks will allow us to design new protocols and contribute to a better understanding of existing ones. This is especially important in areas, such as the Byzantine failure model, where the relationships between different agreement protocols are not well-understood. The method of exploiting Validity to reduce latency, presented in this paper, also appears to be merely an unrelated trick usable with any Consensus protocol. We hope, however, that this and similar ideas will eventually become obvious consequences of some future Consensus framework.
References


