On the Message Complexity of Indulgent Consensus

Seth Gilbert[1], Rachid Guerraoui[1], and Dariusz R. Kowalski[2]

[1] I&C School Of Computer & Communication Sciences
EPFL, 1015 Lausanne, Switzerland
seth.gilbert@epfl.ch, rachid.guerraoui@epfl.ch
[2] Department of Computer Science, University of Liverpool
Liverpool L69 3BX, UK.
d.r.kowalski@csc.liv.ac.uk

Abstract. Many recommend planning for the worst and hoping for the best. In this paper we devise *efficient* indulgent consensus algorithms that can tolerate crash failures and arbitrarily long periods of asynchrony, and yet perform (asymptotically) optimally in well-behaved, synchronous executions with few failures. We present two such algorithms: In synchronous executions, the first has *optimal message complexity*, using only $O(n)$ messages, but runs in superlinear time of $O(n^{1+\varepsilon})$. The second has a message complexity of $O(n \operatorname{polylog}(n))$, but has an optimal running time, completing in $O(f)$ rounds in synchronous executions with at most f failures. Both of these results improve significantly over the most message-efficient of previous indulgent consensus algorithms which have a message complexity of at least $\Omega(n^2)$ in well-behaved executions.

1 Introduction

As in many other fields, it is considered good computing practice to plan for the worst and hope for the best. In the context of distributed computing, this typically translates into devising algorithms that, on the one hand tolerate process failures and arbitrarily long periods of asynchrony, whilst on the other hand, are particularly effective under best-case conditions, namely, few failures and synchrony. Such best-case conditions are usually considered frequent in practice and it makes sense to optimize algorithms with these conditions in mind.

In this paper, we explore this idea in the context of *consensus* [1, 2] in a system of n processes of which a minority can fail by crashing. Given a set of n crash-prone processes, each with initial value v_i; each process needs to *decide* an output satisfying: (1) (agreement) every process decides the same value; (2) (validity) if a process decides value v, then v is the initial value for some process; (3) (termination) every correct process eventually decides.

The question we ask is how "efficient" can a consensus algorithm be when the system is synchronous and $f \leq \lceil n/2 \rceil - 1$ failures actually occur, if the algorithm needs to tolerate arbitrarily long periods of asynchrony. Consensus algorithms that tolerate arbitrarily long periods of asynchrony include [3–8]: they have been called *indulgent* [9]; indulgent consensus is impossible when there are more than a minority of crash failures [3].

	Message Complexity	**Round Complexity**
Alg. 1 (Section 4):	$O(n)$	$O(n^{1+\varepsilon})$
Alg. 2 (Section 5):	$O(n \log^6 n)$	$O(f)$

Fig. 1. Message and round complexity of the two algorithms presented in this paper. Both refer to synchronous executions in which there are no more than $f \leq t$ failures.

Addressing this question requires defining what it means for a consensus algorithm to be "efficient." Usually, this is measured in terms of rounds of communication needed for processes to reach a decision (see, e.g., [10]). There indeed exists an indulgent consensus protocol that reaches a decision in $O(f)$ (in fact, $f + 2$) rounds when the system is synchronous and f processes fail [11]. This algorithm, and in fact all indulgent consensus algorithms that are optimized for synchronous periods (e.g., [4, 8, 12]), exchange $\Omega(n^2)$ messages: all processes send messages to all processes in every round. (In fact, most use at least $\Theta(n^2 f)$ messages.) This pattern of full message exchange is a key subprotocol underlying those algorithms, and is used to detect synchrony and adapt the decision time to the actual number of failures. It is natural to ask whether such a pattern is necessary and whether $\Theta(n^2)$ messages really need to be exchanged.

In other words, is it possible to devise an indulgent consensus protocol that reaches a decision in $O(f)$ rounds when the system is synchronous and no more than f processes fail, while exchanging fewer than $\Theta(n^2)$ messages? If the algorithm does not need to tolerate asynchrony, then the answer is yes [11]: [13] presented a protocol that uses $O((f + 1)n)$ messages and [14] later demonstrated that $O(n + fn^{\epsilon})$ messages are sufficient. However, it is not immediately obvious whether similar results can be achieved if the algorithm must tolerate periods of asynchrony. Clearly, during such a period processes could have divergent views: some may believe the system to be synchronous whereas others may not; some may observe only a small number of failures and hence believe it safe to decide, while others may not. In algorithms with a pattern of full message exchange, these inconsistencies are easy to resolve. The key difficulty in constructing an efficient algorithm that tolerates asynchrony is devising message-efficient techniques for producing a consistent view of the (a)synchrony of the world.

Results

We present in this paper two indulgent consensus algorithms (see Table 1). Both tolerate a minority of the processes failing, and output a decision when the system becomes stable. When the system is synchronous, both algorithms guarantee good performance, both in terms of message-efficiency and round complexity. Each is (asymptotically) optimal in a different sense. The first guarantees optimal message complexity—$O(n)$ messages—in synchronous executions, and terminates in $O(n^{1+\varepsilon})$ rounds. The second is adaptive: it guarantees optimal round complexity—$O(f)$ rounds—in a synchronous execution with no more than $f \leq t$ failures, and has a message complexity of $O(n \log^6 n)$.

The key idea in both algorithms is to simulate an efficient synchronous consensus algorithm, while at the same time detecting asynchrony. If the execution is synchronous, then efficient performance is achieved. If the execution is not synchronous, however, the processes synchronize their view of the world via message-efficient gossip, and eventually fall-back to a less efficient consensus protocol that can better tolerate the uncertain synchrony.

In the case of the second algorithm, which is adaptive, the simulation of the efficient synchronous protocol is more involved: different processes may complete the simulation at different times and (again) with different views of the world. In message-expensive algorithms, this is easy to resolve, as typically all processes decide within one round of each other due to nodes flooding their decision prior to termination. In our case, the combination of adaptivity and possible asynchrony complicated the matters. Throughout the simulation, processes must efficiently determine whether any processes have already produced a decision which is clearly difficult because a process cannot distinguish a failed process from one whose messages are delayed. The solution, again, is through careful use of efficient gossip protocols to synchronize the status of the processes prior to deciding.

Interestingly, both our algorithms can be viewed as generic transformations from synchronous consensus (and gossip) protocols to partially synchronous consensus protocols. Thus future improvements in synchronous algorithms will result immediately in improved indulgent consensus algorithms.

Previous and Related Work

The problem of consensus was first introduced by Pease, Shostak and Lamport [1]. Fisher, Lynch and Paterson [2] showed that consensus is unsolvable in an asynchronous system in which even one process can crash. Thus research on consensus has often focused on synchronous and partially synchronous models of computation. In a seminal paper [3], Dwork, Lynch, and Stockmeyer introduced a model of *eventual synchrony* in which clock skew and message delivery eventually stabilize at some unknown point in the execution. This is the model we adopt in this paper. They showed in [3] that consensus can be solved in the eventually synchronous model if and only if $n \geq 2t + 1$, where t is the tolerable number of crash failures. In [9], Guerraoui coins the term "indulgent" to describe algorithms that can tolerate arbitrarily long periods of asynchrony.

Fisher and Lynch [15] showed that a synchronous solution to consensus requires $t + 1$ rounds, where t is the tolerable number of failures. Dolev and Strong [16] introduce the idea of *early stopping*, or adaptive, consensus protocols, and Lamport and Fischer [17] show that it is possible to terminate in only $f + 2$ rounds in executions with $f < t$ failures. Dolev, Reischuk and Strong [18] show that at least $\min(f + 2, t + 1)$ rounds are necessary. In the context of indulgent consensus, Dutta and Guerraoui [11] show that at least $t + 2$ rounds are required, even in a synchronous execution. There has been a significant amount of recent work on optimizing the running time of consensus in failure-free executions; see, for example [8, 12, 19].

In a synchronous setting, it is relatively straightforward to observe that there is an $\Omega(n)$ lower bound on the message complexity of fault-tolerant synchronous consensus.

Dwork, Halpern and Waarts [20] found a solution with $O(n \log n)$ messages but exponential time. Finally, Galil, Mayer and Yung [14] developed an algorithm with $O(n)$ messages, thus showing that this is the optimal message-complexity. The drawback of their solution is that it runs in superlinear time $O(n^{1+\varepsilon})$, for any fixed $0 < \varepsilon < 1$. Galil, Mayer and Yung [14] also found an adaptive solution with $O(n + fn^{\varepsilon})$ communication complexity, for any $0 < \varepsilon < 1$. Chlebus and Kowalski reduced the number of messages to $O(n \log^2 n)$ for consensus in case $n - t = \Omega(n)$ [21], and recently they developed an adaptive algorithm that tolerates up to $n - 1$ crashes and achieves $O(n \log^5 n)$ message complexity [22]. The message complexity of consensus when no failures actually occur, was studied by Amdur, Weber and Hadzilacos [23] and by Hadzilacos and Halpern [24], and results in the following fact which implies that $O(n)$ message complexity is optimal, regardless of the actual number of failures:

Fact 1 (Amdur, Weber and Hadzilacos [23]) *The message complexity of every (eventually)-synchronous consensus protocol is at least $\Omega(n)$, even in failure-free executions.*

Roadmap

In Section 2 we describe the eventually synchronous model, and in Section 3 we define a series of building blocks, synchronous protocols that will be used in the construction of our algorithms. In Section 4, we describe our first algorithm which guarantees optimal message complexity (in synchronous executions). In Section 5, we describe our second algorithm which is adaptive and guarantees optimal round complexity (in synchronous executions). We outline the proof of correctness in Section 6. In Section 7, we describe instantiations of the building blocks from Section 3, which allows us to analyze the performance of our algorithms in Section 8. We conclude in Section 9.

2 System Model

In this section we describe a basic system model for a *partially synchronous* (or *eventually synchronous*) system, as in [3]. The model is defined by three parameters that are known *a priori*: n, the number of processes, δ, an eventual bound on clock skew, and d, an eventual bound on message delay. There is also a stabilization time, referred to as GST, that is *unknown*. We say that an execution is *synchronous* if GST$= 0$.

In more detail, we consider a system consisting of n message-passing processes, each of which has a unique identifier from the set $[n] = \{1, 2, \ldots, n\}$. Each process is capable of communicating directly with all other processes: prior to GST, messages may be arbitrarily delayed; after GST, every message is delivered within d time. Each process has a local clock, and after GST the clock skew of every process is bounded by δ, i.e., eventually the ratio of the rates of two processes' clocks is at most δ.

We assume that up to $t < \lceil n/2 \rceil$ processes may *crash*, and that processes do not restart or recover. We say that a process is *correct* if it does not crash. We *do not* assume reliable multicast: if a process crashes while sending a message to multiple recipients, then an arbitrary subset of the recipients may receive the message.

We are specifically interested in the performance of algorithms in synchronous executions. We say that an algorithm $\mathcal{A}$ solves consensus *by time* τ in the presence of f

failures if for every synchronous execution with no more than f failures, every node has decided by time τ. We say that algorithm $\mathcal{A}$ has message complexity μ if for every synchronous execution, the total number of messages sent is no more than μ.

3 Building Blocks Protocols

We construct our protocol out of three synchronous building blocks: synchronous consensus, synchronous gossip, and synchronous wake-up. We also use one eventually-synchronous building block, a consensus protocol. In this section, we describe each of these building blocks, and enumerate their properties. In Section 7 we describe how each building block can be implemented from existing protocols.

Synchronous Consensus Protocol. The first basic building block, `SynchConsensus`, is a protocol that solves consensus in synchronous executions. The protocol guarantees the following properties: (1) *Agreement*: In every synchronous execution, all decision values are the same. (2) *Unconditional validity*: In every execution (synchronous or otherwise), every decision is the initial value of some process. (3) *Termination*: In every synchronous execution, every process eventually decides and terminates.

The second property, unconditional validity, is the only guarantee in an execution that is not synchronous. For every $0 \leq f \leq t$, define $\tau^{\texttt{cons}}(f)$ to be the earliest round in which every execution of `SynchConsensus` with no more than f failures terminates. (This is of particular relevance when the consensus protocol is adaptive.)

Synchronous Conditional Gossip. The second building block is a protocol $\texttt{Gossip}(k)$ that solves the conditional gossip problem in synchronous executions with no more than k failures. It is called "conditional" since its guarantees only hold when there are $\leq k$ failures. Each process begins the gossip protocol with a *rumor* v_i. The protocol satisfies the following: (1) *Completion:* In every synchronous execution with at most $k \leq f < n$ failures, every non-failed process eventually receives a rumor from every non-failed process; and (2) *Unconditional validity:* In every execution (synchronous or otherwise), every rumor received is the initial value of some process. For every $0 \leq f \leq t$, define $\tau^{\texttt{gossip}}(f)$ to be the earliest round in which every execution of $\texttt{Gossip}(f)$ with no more than f failures terminates.

Synchronous Conditional Wake-Up. The third building block is a protocol $\texttt{WakeUp}(k)$ that solves the conditional wake-up problem. In conditional wake-up, initially, some subset S of the processes are designated awake, while the rest are designated asleep. The goal of conditional wake-up is that if every process is initially asleep, i.e., $S = \emptyset$, then every process remains asleep and no messages are sent by any process. Conversely, if $S \neq \emptyset$, then every non-failed process wakes up. Again, it is referred to as "conditional" since its guarantees only hold when there are $\leq k$ failures. In more detail, the protocol guarantees the following: (1) *Completion:* In every synchronous execution with at most $k \leq f < n$ failures, if $S \neq \emptyset$, then eventually the protocol terminates and every process concludes that it is awake. (2) *Validity:* In every synchronous execution, if $S = \emptyset$, then every process remains asleep and no messages are sent. For every $0 \leq f \leq t$, define $\tau^{\texttt{wakeup}}(f)$ to be the earliest round in which every execution of $\texttt{WakeUp}(f)$ with no more than f failures terminates.

Partially-Synchronous Consensus Protocol. The final building block is an arbitrary eventually-synchronous consensus protocol `PartSynchConsensus`; it guarantees the usual properties of consensus: agreement, validity, and termination. There are a variety of protocols that satisfy these requirements, including, for example, [3, 5].

4 Indulgent Consensus

In this section we present our first indulgent consensus protocol. When instantiated using the appropriate building-block protocols, the result is an (asymptotically) message-optimal algorithm. (See Theorem 3.) The main idea is to first simulate an efficient *synchronous* consensus protocol `SynchConsensus` (see Section 7.1), and then determine whether it has completed successfully. If so, then each process can decide that value and terminate; otherwise processes run a fall-back partially synchronous consensus protocol that is not as message efficient. The main difficulty, then, is correctly detecting when an execution is synchronous without sending too many messages.

4.1 Simulating Synchronous Rounds

Each process simulates synchronous rounds in the standard manner based on message delay d and clock skew δ. Recall that in a synchronous execution, at time τ the clock at every process i is in the range $[(1-\delta)\tau, (1+\delta)\tau]$. Let $\rho = (1+\delta)/(1-\delta)$. The first simulated round r_1 ends at time $d/(1-\delta)$ according to the local clock at each process. Simulated round r ends for each process at time: $\tau^{\mathtt{sim}}(r) = \frac{d}{1-\delta}\sum_{j=0}^{r-1}\rho^j$ according to the local clock of that process. In a synchronous execution, every message sent at the beginning of round r according to the local clock of the sending process is received by the end of round r according to the local clock of the receiving process.

4.2 Protocol Description

The protocol proceeds in four phases: (1) Agreement Phase, (2) Locking Phase, (3) Decision Phase, (4) Fall-back Phase. When the protocol begins, the proposal for process i is stored in e_i, its estimate. Process i also maintains a variable $status_i$ that indicates its current status. Initially $status_i =$ proposal, indicating that the estimate is the initial value. As the status advances during the protocol to higher levels, it never returns to a lower level.

1. Agreement Phase. In the first phase, the processes together simulate the consensus protocol `SynchConsensus` for $\tau^{\mathtt{cons}}(t)$ rounds. If the execution is, in fact, synchronous, then for each correct process the consensus simulation will output a decision; all such decisions will agree. If the execution is not synchronous, then the consensus protocol may not terminate, or may output different decisions at different processes. If a process discovers that its simulation reaches a decision, then this decision is stored as its estimate e_i, and its status is advanced to candidate. Notice that processes do not decide on the value output by `SynchConsensus` at this time.

The agreement phase continues until $\tau^{\mathtt{cons}}(t)$ rounds have been simulated (where $t < \lceil n/2 \rceil$ is the maximum tolerated number of failures). If the simulated consensus

protocol SynchConsensus has not terminated, then the simulation is halted. In this case, any process that has not decided will (eventually) enter the fall-back phase.

2. Locking Phase. In the second phase, the processes together simulate the synchronous conditional gossip protocol $\texttt{Gossip}(t)$ for $\tau^{\texttt{gossip}}(t)$ rounds. Each process i uses e_i and $status_i$ as its initial rumor. Thus, in a synchronous execution, every non-failed process receives the rumors of all other non-failed processes. At the end of the phase, process i advances its status under the following conditions: (1) it has received a rumor from at least $\lfloor n/2 \rfloor + 1$ processes that have a status of candidate, locked, or decided for some value v; (2) estimate $e_i = v$; and (3) $status_i =$ proposal or candidate. In this case, process i updates its status to locked. We will argue (Lemma 1) that at most one value is locked in an execution.

3. Decision Phase. In the third phase, the processes repeat the (synchronous) conditional gossip protocol $\texttt{Gossip}(t)$ for $\tau^{\texttt{gossip}}(t)$ further (synchronous) rounds. Each process i again uses e_i and $status_i$ as its initial rumor. At the end of the third phase, process i advances its status under the following conditions: (1) it receives a rumor from at least $\lfloor n/2 \rfloor + 1$ processes that have a status of locked or decided for same value v; and (2) estimate $e_i = v$; and $status_i =$ locked. In this case, process i updates $status_i =$ decided, and decides e_i. If all the processes have decided, then from this point on no further messages are broadcast, and the protocol is considered to be terminated.

4. Fall-back Phase. In the final phase, if any process has not yet decided, then the processes all resort to the fall-back consensus protocol PartSynchConsensus. This phase occurs only in executions that are not synchronous. The synchronous round simulation is abandoned at this point.

The first step in the fall-back phase is to collect the final status of all the other processes. This proceeds as follows: (1) Each process i that has not yet decided in the previous phase sends a fall-back message to every other process: $\langle \mathsf{fallback}, e_i, status_i \rangle$. (2) Any process that receives a fall-back message enters the fall-back phase and, if it has not already done so, immediately sends a fall-back message $\langle \mathsf{fallback}, e_j, status_j \rangle$ to every process, even if it has previuosly decided. (3) When a process i receives $\lfloor n/2 \rfloor + 1$ fall-back messages, it determines if there are any locked values. That is, if any fall-back message contains status locked, then process i sets its estimate e_i to the value of that message.

Since every message is eventually delivered, and since a majority of processes are correct, it is easy to see that if any correct process begins the fall-back phase, then eventually every process receives $\lfloor n/2 \rfloor + 1$ fall-back messages. Thus, if any value has been locked by a majority of the processes during the initial three phases, then each process executing the fall-back phase will adopt that value as its estimate. Every process that has received a fall-back message then executes PartSynchConsensus, where process i uses estimate e_i as its proposal. Eventually PartSynchConsensus produces a decision, and each process decides this value and terminates.

5 Adaptive Indulgent Consensus

In this section we show how to modify the protocol presented in Section 4 to develop an *adaptive* indulgent consensus protocol. Recall that the protocol in Section 4 begins by simulating the consensus protocol `SynchConsensus` in the agreement phase. If `SynchConsensus` is adaptive, it terminates early in synchronous executions with few failures. The goal of this section is to detect when the `SynchConsensus` simulation has terminated. This detection is accomplished by pausing the consensus simulation every so often and executing a variant of the locking and decision phases, using conditional gossip primitives designed for $f \leq t$ failures. Before resuming the consensus protocol simulation, we execute a conditional wake-up protocol: if some processes have decided and other have not yet decided, this protocol wakes the processes that have already decided so that they can continue with the (simulated) consensus protocol.

We divide the agreement phase into $\log n$ epochs numbered from 0 to $\log n - 1$. Epoch x has length $O(2^x)$, and simulates $O(2^x)$ rounds of `SynchConsensus`. The epochs are structured such that by the end of epoch x, the system has finished executing round $\tau^{\texttt{cons}}(2^x)$ of `SynchConsensus`; thus in a synchronous execution with $\leq 2^x$ failures, `SynchConsensus` completes by the end of epoch x. (Notice that there are $< 2^{\log n - 1} = n/2$ failures.)

In more detail, each epoch consists of four phases: (1) *Wake-up:* waking the processes that have already decided; (2) *Agreement:* simulating some rounds of the consensus protocol; (3) *Locking:* execute conditional gossip and determine if any of the values can be locked, and (4) *Deciding:* execute conditional gossip and determine if any of the values can be decided. If, at the end of $\log n$ epochs a process has not yet decided, then it enters the fall-back phase.

1. Waking the Processes. At the beginning of epoch x, there are three possibilities: all the processes have decided, none have decided, or some have decided and some have not. In this last case, a problem might occur if some processes have decided, and thus stopped participating voluntarily in future epochs, while others have not yet decided and need to continue the protocol. The first step in epoch x, then, is to execute `WakeUp`(2^x): each process that has decided is initially asleep and each process that is undecided is initially awake. This step takes $\tau^{\texttt{wakeup}}(2^x)$ rounds, and guarantees that if the execution is synchronous and there are no more than 2^x failures, then every process is awake.

2. Agreement. The second step in epoch x is to simulate some rounds of the consensus protocol `SynchConsensus`, continuing from the last round simulated in the previous epoch. In epoch 0, the processes simulate the first $\tau^{\texttt{cons}}(1)$ rounds of the protocol. In epoch $x > 0$, the process simulate rounds $\tau^{\texttt{cons}}(2^{x-1}) + 1, \ldots, \tau^{\texttt{cons}}(2^x)$ of the consensus protocol. If a process has decided in an earlier epoch, then it continues to execute the simulation of `SynchConsensus` only if it was awoken in the wake-up step, and if there are further rounds to simulate.

As in Section 4, each process i maintains two variables: e_i, its estimate, and $status_i$, its status. Initially, e_i is process i's proposal, and $status_i = \mathsf{proposal}$. If process i discovers that its simulated consensus protocol has decided value v, and if process i has status equal to proposal, then process i sets $e_i = v$ and advances $status_i = \mathsf{candidate}$.

If an execution is synchronous and has fewer than 2^x failures, then the simulated consensus protocol will terminate for all non-failed processes by the end of epoch x.

3. Locking. The third step in epoch x is to simulate the conditional gossip protocol $\texttt{Gossip}(2^x)$ for $\tau^{\texttt{gossip}}(2^x)$ (simulated) rounds, with e_i and $status_i$ as the rumor for process i. This step is equivalent to the locking phase described in Section 4, except that $\texttt{Gossip}(2^x)$ is executed, instead of $\texttt{Gossip}(t)$. If at the end of the locking phase, process i has received rumors from at least $\lfloor n/2 \rfloor + 1$ processes that all have value e_i as a candidate, locked, or decided, and if $status_i =$ proposal or candidate, then process i locks value e_i. If a process has decided in an earlier epoch, then it executes the gossip only if it was awoken in the wake-up step; otherwise, it remains silent.

4. Deciding. The fourth step in epoch x is to again together simulate $\texttt{Gossip}(2^x)$ for $\tau^{\texttt{gossip}}(2^x)$ (simulated) rounds, again with e_i and $status_i$ as the rumor for process i. This step is equivalent to the deciding phase described in Section 4, except that $\texttt{Gossip}(2^x)$ is executed, instead of $\texttt{Gossip}(t)$. If at the end of the deciding phase, process i has received rumors from at least $\lfloor n/2 \rfloor + 1$ processes that have all locked or decided value e_i, and if $status_i \neq$ decided, then process i decides value v. As in the previous step, if a process has decided in an earlier epoch, then it executes the gossip protocol only if was awoken in the wake-up step; otherwise, it remains silent.

Fall Back. If, at the end of all $\log n$ epochs, any process has not yet decided, then it enters the fall-back phase, as described in Section 4, sending and collecting fall-back messages, and running `PartSynchConsensus`.

6 Analysis

In this section we provide an outline of the proof that the protocol presented in Section 5 guarantees agreement, validity, and termination. Performance results are given in Section 8. We begin by showing that in every execution, there is at most one value that is decided. The key lemma, in this case, is that at most one value is locked during a locking phase. Notice that this does not depend in any way on the agreement property of `SynchConsensus` which only holds in synchronous executions.

Lemma 1. *In every execution, there is at most one value v such that $e_i = v$ and $status_i =$ locked for any i.*

Proof. Assume for the sake of contradiction that i and j have locked two distinct values v and v' (possibly in two different epochs). This implies that each received rumors during a locking phase from a majority of processes indicating that v and v', respectively, were candidate, locked, or decided values. Thus, some process k (in the intersection of the two majorities) must have at one point had value v as a candidate, locked, or decided value and at another point value v' as a candidate, locked, or decided value. But a process never changes its estimate after it has become a candidate, implying a contradiction.

Lemma 2. *In every execution, there is at most one value v that is decided.*

Proof. Suppose for contradiction that processes i and j decide two different values v and v'. There are three cases: (Case 1) Both decide prior to the fall-back phase: This contradicts Lemma 1, as prior to the fall-back phase, a process only decides a value that has been previously locked. (Case 2) Both decide during the fall-back phase: This contradicts the agreement property of `PartSynchConsensus`, which guarantees that at most one value is decided. (Case 3) One (say, i) decides v prior to the fall-back phase and one (say, j) decides v' during the fall-back phase: We argue that every process k begins the fall-back phase with initial value v. Process i decides prior to the fall-back phase only if it receives gossip messages indicating that a majority of processes have locked value v. In the first step of the fall-back phase, process k receives fall-back messages from a majority of the processes. Since a process never changes its estimate once it is locked (prior to the fall-back phase), we can conclude that process k receives a message indicating that value v has been locked. Since there is at most one locked value, by Lemma 1, we conclude that process k adopts value v as its proposal in the fall-back phase. Since every process proposes value v in the fall-back phase, the validity of `PartSynchConsensus` implies that every non-failed process decides v, resulting in a contradiction.

Next, it follows immediately from the unconditional validity of `SynchConsensus` and `Gossip`(t), and from the validity of `PartSynchConsensus`, that the decision is valid:

Lemma 3. *If v is decided in some execution, then for some process i, initially $e_i = v$.*

Finally, it is easy to see that, due to the fall-back protocol, the protocol eventually terminates in all executions:

Lemma 4. *In all executions, every process eventually decides and stops sending messages.*

7 Implementing the Three Synchronous Building Blocks

In this section we describe efficient implementations of the building-block protocols described in Section 3.

7.1 Implementing `SynchConsensus`

This section describes two synchronous consensus protocols, both derived from prior work. The first is adaptive and uses $O(n \log^6 n)$ messages; the second uses a superlinear number of rounds but has optimal $O(n)$ message complexity.

Adaptive Synchronous Consensus. In this section, we outline the construction of an adaptive synchronous consensus protocol that is message efficient. We proceed in three steps: we start with a synchronous binary consensus developed in [22]; then we construct a multivalue consensus protocol; finally we transform the resulting protocol into an *adaptive* protocol. In each step, the challenge is to not increase the asymptotic running time and message complexity too much.

Efficient binary synchronous consensus. In [22], Chlebus and Kowalski introduce a binary, message-efficient consensus protocol that tolerates up to $n-1$ failures, decides in time $O(n)$ and sends $O(n \log^5 n)$ point-to-point messages.

From binary to multivalue consensus. While the protocol presented in [22] is for binary consensus, it can be readily modified to efficiently support multivalue consensus. Typically, binary consensus protocols are translated into multivalue consensus protocols by agreeing on each bit one at a time. In order to achieve unconditional validity and to avoid increasing time complexity above $\Theta(n)$, a slightly different approach is needed. We construct a binary tournament tree and use binary consensus to navigate the tree. This results in a synchronous multivalue consensus protocol that runs in $O(n)$ times and $O(n \log^6 n)$ message complexity.

Adaptive synchronous consensus Chlebus and Kowalski show in [22] how to transform a message-efficient, synchronous consensus protocol into an *adaptive* message-efficient synchronous consensus protocol, with an (additive) additional $O(n \log^4 n)$ message complexity. The end result is a synchronous, adaptive, message-efficient, that is having $O(n \log^6 n)$ message complexity, consensus protocol `SynchConsensus` that guarantees unconditional validity:

Proposition 1. *There exists a synchronous multivalue consensus protocol with message complexity $O(n \log^6 n)$ and round complexity $O(f)$ in executions with $\leq f$ failures.*

Message-Optimal Synchronous Consensus. In this section we outline the construction of a message-optimal synchronous consensus protocol that uses only $O(n)$ messages and runs in times $O(n^{1+\varepsilon})$ for every $0 < \varepsilon < 1$. We begin by describing a protocol that solves the Interactive Consistency problem, a stronger variant of consensus in which processes agree not simply on a single value, but rather on a vector of decision values, including one for each correct process[3]. Formally, the IC problem is defined as follows: each process i begins with an initial value v_i, and outputs a decision vector D_i such that the following properties are satisfied: (1) *Agreement*: In every synchronous execution, the decision vector D_i of all processes is the same. (2) *Unconditional validity*: In every execution (synchronous or otherwise), if D_i is the decision vector of process i, then $D[j]_i$ is either the initial value of process j or $\perp$. (3) *Conditional validity*: If the execution is synchronous and j is correct, then $D[j]_i \neq \perp$. (4) *Termination*: Eventually every process outputs a decision vector D_i and terminates.

In [14], there is a synchronous protocol that efficiently solves the *checkpoint* problem, a variant of IC. In particular, the checkpoint problem requires each process i to output a set of processes P_i (rather than a set of values) where every correct process is in the set P_i, and every process in P_i is non-failed at the beginning of the execution. We claim that every synchronous checkpoint protocol can be transformed into a synchronous algorithm for IC:

Lemma 5. *If A solves synchronous checkpoint in τ rounds with message complexity μ, then there exists a synchronous protocol $\mathcal{A}'$ that solves IC in τ rounds with message complexity μ.*

[3] It is interesting to notice that IC cannot be solved in a partially synchronous model [9]. We depend on the IC protocol only in synchronous executions, and hence there is no contradiction.

We conclude from Lemma 5, along with the checkpoint protocol from [14]:

Proposition 2. *There exists a synchronous multivalue consensus protocol with message complexity $O(n)$ and round complexity $O(n^{1+\varepsilon})$, for any $0 < \varepsilon < 1$.*

7.2 Implementing `Gossip`(k)

In this section, we describe two synchronous (conditional) gossip protocols. The first terminates in $O(k)$ rounds and uses $O(n \log^4 n)$ messages, while the second uses a superlinear number of rounds but has $O(n)$ message complexity.

Adaptive Conditional Gossip. In [22], Chlebus and Kowalski present a gossip protocol tolerating up to $n - 1$ failures that has message complexity $O(n \log^4 n)$ and completes in $O(\log^3 n)$ rounds. When $k \geq \log^3 n$, the running time is $O(k)$, as desired. When $k \leq \log^3 n$, we resort to a simpler two-round protocol in order to guarantee termination time $O(k)$: in the first round, each process sends its rumor to processes $[1, \ldots, k+1]$; in the second round, processes $[1, \ldots, k+1]$ send all the received rumors to all the other processes. Notice that this satisfies the conditional completion property, as there are at most k failures, and is message efficient, as it requires at most $2(k+1)n = O(n \log^3 n)$ messages when $k < \log^3 n$.

Proposition 3. *For all $f < n$, there exists a synchronous conditional gossip protocol with message complexity $O(n \log^4 n)$ and round complexity $O(f)$ in executions with at most f failures.*

Message-Efficient Conditional Gossip. Recall from Section 7.1, there exists a protocol solving Interactive Consistency in $O(n^{1+\varepsilon})$ rounds with $O(n)$ messages. Notice that any solution to Interactive Consistency is also a solution to gossip, as each process outputs a set of initial values from every correct process. We thus conclude:

Proposition 4. *There exists a synchronous conditional gossip protocol with message complexity $O(n)$ and round complexity $O(n^{1+\varepsilon})$ in executions, for any $0 < \varepsilon < 1$.*

7.3 Implementing `WakeUp`(k)

The conditional wake-up problem is quite close to the conditional gossip problem; the primary difference is that processes initially designated to be asleep must not send any messages at least until they have received a message from a process that was initially awake. Thus, from the point of view of the gossip algorithm, a sleeping process can be treated as faulty until it is awoken. Thus the wake-up problem can be solved using any synchronous gossip protocol that satisfies the following additional *Polling Property*: in every execution, for every faulty process i there is some process j that, prior to failing, sends a message to i. Nearly every "reasonable" gossip protocol, including the one described in Section Section 7.2, has this property. The simple two-round protocol (when $k \leq \log^3 n$)) also clearly has this property. We conclude:

Proposition 5. *For all $f \leq t$, there exists a synchronous conditional wake-up protocol that has message complexity $O(n \log^4 n)$ and round complexity $O(f)$ in executions with at most f failures.*

8 Performance Analysis

In this section, we analyze the efficiency of the two algorithms. We begin with the adaptive protocol from Section 5 where `SynchConsensus` is instantiated by the consensus protocol posited by Proposition 1, `Gossip`(k) is instantiated by the gossip protocol posited by Proposition 3, and `WakeUp`(k) is instantiated by the wake-up protocol posited by Proposition 5.

Lemma 6. *For every synchronous execution with no more than $f \leq t$ failures, every process decides by time $O(f)$, terminating prior to the beginning of the fall-back phase.*

Proof. If $f = 1$, consider epoch $x = 0$; otherwise, consider epoch x such that $2^{x-1} < f \leq 2^x$. There are two possibilities at the beginning of epoch x: either some process has already decided in an earlier epoch, or no process has decided in an earlier epoch. By the conditional guarantee of the wake-up protocol, however, in either case every non-failed process awakes to participate in epoch x.

Next, by the adaptivity property of `SynchConsensus`, we can conclude that the simulated consensus protocol has output a decision at each non-failed process by the end of the agreement step of epoch x. Thus every non-failed process has status either a candidate, locked, or decided. Since the simulated consensus protocol guarantees agreement, every process with status candidate has the same value. Since every value that is locked or decided was previously a candidate, we can conclude that every process in fact has the same value.

In the locking step of epoch x, since there are no more than 2^x failures, the conditional gossip ensures that each non-failed process receives rumors from a majority of processes, all of which have value v as candidate, locked, or decided. We can thus conclude that at the end of the locking step, every non-failed process has either locked or decided value v. Similarly, in the decision step of epoch x, since there are no more than 2^x failures, the conditional gossip ensures that each non-failed process receives rumors from a majority of processes. We can thus conclude that at the end of the decision phase, every process has decided value v. From this point on, no process sends any further messages. Thus we conclude that by the end of epoch x, every non-failed process has terminated.

Finally, we calculate the total running time through the end of epoch x. First, simulating `SynchConsensus` through the end of epoch x requires $O(\tau^{\texttt{cons}}(2^x))$ rounds, which by the choice of `SynchConsensus` is $O(2^x)$ rounds. Next, notice that for every epoch $y \leq x$, each process executes two instances of `Gossip`(2^y) and one instance of `WakeUp`(2^y); these instances take time $O(\tau^{\texttt{gossip}}(2^y))$ and $O(\tau^{\texttt{wakeup}}(2^y))$, respectively, which are both, by assumption, $O(2^y)$. Thus, for each epoch y, the wake-up, locking, and decision phases cost $O(2^y)$ rounds, and hence when summed from epoch 0 to epoch x result in a running time of $O(2^x)$ rounds. Thus the total running time to the end of epoch x, in terms of synchronous rounds, is $O(2^x) = O(f)$, implying a termination time of $\tau^{\texttt{sim}}(O(f)) = O(f)$.

We next argue that the resulting protocol is message efficient:

Lemma 7. *In every synchronous execution, the processes send $O(n \log^6 n)$ messages.*

Proof. During the entire simulation of `SynchConsensus`, the processes collectively send $O(n \log^6 n)$ messages. In each epoch x, each (non-failed) process executes two instances of `Gossip`(x) and one instance of `WakeUp`(x); each such instance uses $O(n \log^4 n)$ messages, resulting in $O(n \log^6 n)$ messages total. By Lemma 6, we conclude that each process decides no later than the final epoch, as desired.

Thus we conclude:

Theorem 2. *There exists an adaptive indulgent consensus protocol with message complexity $\Theta(n \log^6 n)$ and running time $O(f)$ in synchronous executions with no more than f failures.*

We next briefly examine the performance of the protocol presented in Section 4, where `SynchConsensus` is instantiated by the protocol posited by Proposition 2 and `Gossip`(k) is instantiated by the gossip protocol posited by Proposition 4. Since the structure of the protocol is identical to that of one epoch of the adaptive protocol, we conclude (much as in Section 6, and omitted to avoid redundancy and save space) that the protocol solves the gossip problem and eventually terminates:

Theorem 3. *There exists an indulgent consensus protocol with message complexity $\Theta(n)$ and a running time $O(n^{1+\varepsilon})$ in synchronous executions.*

9 Discussion and Open Questions

We have shown how to implement efficient indulgent consensus algorithms in an eventually-synchronous network. In fact, the algorithms described, with minor modifications, tolerate an even less well-behaved environment. First, even if messages are lost prior to GST, both algorithms continue to behave correctly, as long as each process that has entered the fall-back phase repeats its fall-back message until a decision is reached. Second, even if the bounds d and δ are incorrect, both algorithms continue to solve consensus as long as the network eventually stabilizes for some (unknown) $\hat{d}$ and $\hat{\delta}$; only the message-efficiency is sacrificed. Third, in synchronous executions both algorithms can tolerate more failures, in fact, up to $n - 1$ failures, as long as no more than a minority fail in executions that are not synchronous.

One major open question raised by this paper is whether there exists a protocol that is both optimal in message complexity and linear in round complexity. The answer is unknown even for synchronous networks. Another question is whether it is possible to achieve better message complexity in an adaptive algorithm. For some values of f (when f is much smaller than n), it is possible to use an alternative instantiation of the building blocks derived from [14] to achieve a somewhat better message complexity, while terminating in $O(f)$ time.

References

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

2. Fisher, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2) (1985) 374–382
3. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2) (1988) 288–323
4. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
5. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16(2)** (1998) 133–169
6. Mostefaoui, A., Raynal, M.: Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In: Proceedings of the 13th International Symposium on Distributed Computing (DISC). (1999) 49–63
7. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Transactions on Computers **53**(4) (2004) 453–466
8. Schiper, A.: Early consensus in an asynchronous system with a weak failure detector. Distributed Computing **10**(3) (1997) 149–157
9. Guerraoui, R.: Indulgent algorithms (preliminary version). In: Proceedings of the 19th Symposium on Principles of Distributed Computing (PODC). (2000) 289–297
10. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
11. Dutta, P., Guerraoui, R.: The inherent price of indulgence. In: Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC). (2002) 88–97
12. Lamport, L.: Fast paxos. Technical Report MSR-TR-2005-12, Microsoft (2005)
13. Chandra, T., Toueg, S.: Time and message efficient reliable broadcasts. In: Proceedings of the 4th International Workshop on Distributed Algorithms (WDAG). (1990) 289–303
14. Galil, Z., Mayer, A., Yung, M.: Resolving message complexity of byzantine agreement and beyond. In: Proceedings of the 36th Symposium on Foundations of Computer Science (FOCS). (1995) 724–733
15. Fisher, M., Lynch, N.: A lower bound for the time to assure interactive consistency. Information Processing Letters (IPL) **14**(4) (1982) 183–186
16. Dolev, D., Strong, H.: Requirements for agreement in a distributed system. Technical Report RJ 3418, IBM Research, San Jose, CA (March 1982)
17. Lamport, L., Fisher, M.: Byzantine generals and transaction commit protocols. Unpublished (April 1982)
18. Dolev, D., Reischuk, R., Strong, H.R.: Early stopping in byzantine agreement. Journal of the ACM **37**(4) (1990) 720–741
19. Charron-Bost, B., Schiper, A.: Improving Fast Paxos: being optimistic with no overhead. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC). (2006) 287–295
20. Dwork, C., Halpern, J., Waarts, O.: Performing work efficiently in the presence of faults. SIAM Journal on Computing **27**(5) (1998) 1457–1491
21. Chlebus, B., Kowalski, D.: Gossiping to reach consensus. In: Proceedings of 14th Symposium on Parallel Algorithms and Architectures (SPAA). (2002) 220–229
22. Chlebus, B., Kowalski, D.: Robust gossiping with an application to consensus. Journal of Computer and System Science **72**(8) (2006) 1262–1281
23. Amdur, S., Weber, S., Hadzilacos, V.: On the message complexity of binary agreement under crash failures. Distributed Computing **5**(4) (1992) 175–186
24. Hadzilacos, V., Halpern, J.: Message-optimal protocols for byzantine agreement. Mathematical Systems Theory **26**(1) (1993) 41–102