

The Mutable Consensus Protocol*

J. Pereira
U. Minho
jop@di.uminho.pt

R. Oliveira
U. Minho
rco@di.uminho.pt

Dep. de Informática, Campus de Gualtar
4710-057 Braga, PORTUGAL

Abstract

In this paper we propose the *mutable consensus protocol*, a pragmatic and theoretically appealing approach to enhance the performance of distributed consensus. First, an apparently inefficient protocol is developed using the simple *stubborn channel* abstraction for unreliable message passing. Then, performance is improved by introducing judiciously chosen finite delays in the implementation of channels. Although this does not affect correctness, which rests on an asynchronous system model, it makes it likely that the transmission of some messages is avoided and thus the message exchange pattern at the network level changes noticeably. By choosing different delays in the underlying stubborn channels, the mutable consensus protocol can actually be made to resemble several different protocols.

Besides presenting the mutable consensus protocol and four different mutations, we evaluate in detail the particularly interesting *permutation gossip* mutation, which allows the protocol to scale gracefully to a large number of processes by balancing the number of messages to be handled by each process with the number of communication steps required to decide. The evaluation is performed using a realistic simulation model which accurately reproduces resource consumption in real systems.

1 Introduction

Several distributed programming problems such as atomic broadcast, view synchrony and atomic commitment can be reduced to consensus [2], hence the relevance of correct and efficient consensus protocols. Nevertheless, a fundamental result states the impossibility of deterministic consensus in asynchronous distributed systems where at least one process may crash [3]. This impossibility can be circumvented by strengthening the asynchronous model with additional assumptions.

*This report is an extended version of [1].

We focus on protocols based on unreliable failure detectors [4, 5] in asynchronous message passing systems where processes may fail by crashing. These protocols execute in asynchronous rounds with a rotating coordinator. In each round, an estimate is broadcast to all participants by the coordinator of the round. The protocol then tries to gather a majority of votes, either to decide or to enter the next round. When a value is decided it is reliably broadcast to all participants.

These protocols differ mostly on how votes are collected. As an example, in the early consensus protocol [5] all votes are broadcast to the entire set of participants, leading to a quadratic number of messages and imposing a heavy load on the network. With a centralized protocol [4], votes are collected by the coordinator and relayed to other participants only after a decision has been reached, reducing the load on the network at the expense of an additional communication step. Such differences have a definite impact in performance measured in realistic settings [6].

In this paper we address this issue: The trade-off between the number of communication steps and the number of messages transmitted and handled by processes. This trade-off depends on the relative cost of sending, transmitting and receiving messages, as well as, on the availability of processing and network resources. The evaluation of the proposed protocol is therefore done in a realistic environment, ensuring that in practice the proposal algorithm results in good performance. By varying system parameters we ensure that performance gains can be obtained across a variety of concrete environments.

Our proposal is done in two steps. First, we present a new consensus protocol based on *stubborn channels* [7]. Although the result is apparently not attractive by any performance metric, especially when considering the number of messages exchanged, we notice an interesting property: As messages can be lost by stubborn channels, it is possible that only a small fraction of the messages sent by the protocol are actually transmitted through the underlying network. In fact, we can easily fabricate valid runs which exchange a much lower number of messages at the network level. Unfortunately, it is highly unlikely that a naive implementation produces such desirable runs.

We therefore seek an implementation which maximizes the likelihood of desirable runs. Interestingly, this can be achieved simply by introducing finite delays in a naive implementation of stubborn channels. Moreover, as delays are finite this does not in any way compromise the correctness of the protocol, which assumes an asynchronous system model. In practice, judiciously chosen delays make it likely that only desirable runs occur thus resulting in very good performance in practical metrics such as the latency and the number of bytes transmitted.

Such delays avoid actual transmission of messages due to two different reasons. The first is that they increase the likelihood of a more recent message being sent in the meantime, which in stubborn channels discards all previously sent messages. The second is that if a decision can be reached by all processes before the delay expires, the transmission can be entirely avoided

in practice. As an example, consider the usage of consensus to implement view synchronous multicast, in which an instance of consensus is run to install each view [2]. As soon as a process has started receiving messages (or acknowledgments to messages) from all others in the recently installed view, it knows that all participants in the previous consensus instance have decided. It may therefore terminate the consensus protocol and flush all pending messages.

Different configurations of delays lead to different messages being actually transmitted and thus result in different classes of desirable runs. Some of these resemble the message exchange pattern of well known protocols. Others result in innovative message exchange patterns with desirable performance characteristics. We therefore call it the *mutable consensus protocol* and each of the combinations of the protocol with an implementation of stubborn channels a *protocol mutation*. In this paper we introduce four distinct mutations. Two of them mimic well known protocols [4, 5]. A third is called the *ring* and uses very little resources at the expense of high latency. Finally, the *permutation gossip* mutation allows the protocol to scale to very large groups.

The paper is structured as follows: Section 2 motivates the work by introducing the consensus problem. Section 3 presents the mutable consensus protocol based on stubborn channels. Section 4 introduces protocol mutations. Section 5 briefly compares their performance. The *permutation gossip* mutation is examined in detail in Section 6. Section 7 briefly discusses related work and Section 8 concludes the paper.

2 Background

In this section we motivate our work by introducing the consensus problem and its applications in fault-tolerant distributed systems, and the difficulty in obtaining efficient implementations of existing protocols. We start by briefly describing the system model assumed.

2.1 System Model

We consider an asynchronous, message-passing system consisting of a finite set of processes $\{p_1, p_2, \dots, p_n\}$. There is no global clock but each process p_i has access to a local monotonically increasing clock that can be read in variable Clock_i .

Processes may only fail by crashing, and once a process crashes it does not recover. A process that does not crash is said *correct* and we assume that a majority of the processes are correct. Our model of computation is augmented with a failure detector oracle of class $\diamond S$ [4] enabling us to circumvent the FLP impossibility result [3].

Processes are completely connected through a set of fair-lossy communication channels [8]. Each channel connecting process p_i to p_j is defined by a pair of primitives $\text{Send}_{i,j}(m)$ and

$\text{Receive}_{j,i}(m)$. Such a channel is a reasonable abstraction of the service provided by existing connectionless network layers and basically ensures that no spurious messages are created, message duplication is finite, and that each message has a non-null probability of being delivered. A reliable channel can be reduced to a fair-lossy channel by buffering and retransmitting messages [8].

2.2 Consensus

The consensus problem abstracts agreement in fault-tolerant distributed systems, in which a set of processes agree on a common value despite starting with different opinions. More formally, all processes are expected to start the protocol proposing some value through function `Consensus` and then decide on its return value such that the following properties hold [4]:

- *Validity*: If a process decides v , then v was proposed by some process.
- *Agreement*: No two processes decide differently.
- *Termination*: Every correct process eventually decides some value.

We focus on consensus protocols based on unreliable failure detectors of class $\diamond S$ and using reliable channels [4, 5, 9]. These protocols execute in asynchronous rounds with a rotating coordinator. In each round, an estimate is broadcast to all participants by the coordinator of the round. The protocol then tries to gather a majority of votes to decide. Whenever a majority of processes votes favorably in a round, the decision is said to be locked, as no other value can then be decided. When a process decides, it relays the decision to all participants.

The failure detector is used to avoid blocking when the coordinator of the round has crashed. Whenever the coordinator is suspected processes try to leave the current round to force the coordinator to change. Before an estimate can be broadcast by the new coordinator, it is required that it gathers also a majority of votes from participating processes. This enforces agreement by ensuring that after a value has been locked it will be used as the estimate for all future rounds. In fact, such protocols have the desirable property of being indulgent [10]. Even if the detector misbehaves (*i.e.*, the assumption that the failure detector is of class $\diamond S$ turns out to be wrong) the protocol ensures safety properties.

Protocols differ mostly in how majorities of votes are collected. In a centralized protocol [4], all votes are gathered by the current round coordinator. In detail, when entering a round the coordinator collects estimates from the previous round. Then it broadcasts a selected estimate and collects the acknowledgments. Upon receiving acknowledgments from a majority, the decision is broadcast. This allows the decision to be reached in three communication steps and requires that only the coordinator handles messages from all participants.

On the other hand, in a decentralized protocol [5] all votes are broadcast to all participants, making it possible that each process independently gathers a majority and thus reaches a decision. This allows the decision to be reached in two communication steps at the expense of a larger number of messages exchanged. The number of messages exchanged can be reduced by using broadcast mechanisms at the network level when available, but still requires that all participants handle messages from all others.

It has been shown that these two protocols are extreme instantiations of a more general protocol [9]. In between, innovative protocols in which a subset of processes gather votes can be obtained. Nevertheless, there are always processes which must receive and then send messages to all others. This is unfortunate as, in practice, the performance of a distributed protocol in general, and in particular its scalability to large numbers of participants, is tightly related to the number of messages sent and received by each process. The available processing power of such participants thus directly translates to an upper bound on the scalability of protocols. The best trade-off depends on the overhead associated with transmitting and handling messages in a concrete setting.

Limitations on scalability can usually be mitigated by distributing the message load by all processes. Nevertheless, for each new protocol which uses an innovative message exchange pattern that is suited to a particular environment one would have to redo correctness proofs. The added complexity of implementing such protocols would also require additional effort to ensure that the implementation itself is correct.

3 Mutable Consensus Protocol

In this section we introduce the mutable consensus protocol. We start by presenting the definition and a simple implementation of stubborn channels [7].

3.1 Stubborn Channels

A stubborn channel [7] connecting two processes p_i and p_j is an unreliable channel defined by a pair of primitives $\text{sSend}_{i,j}(m)$ and $\text{sReceive}_{j,i}(m)$, that satisfy the following two properties:

- *No-Creation*: If p_i receives a message m from p_j , then p_j has previously sent m to p_i .
- *Stubborn*: Let p_i and p_j be correct. If p_i sends a message m to p_j and p_i indefinitely delays sending any further message to p_j , then p_j eventually receives m .

A stubborn channel is easily implementable over a fair-lossy channel: It suffices to buffer the last message sent and periodically retransmit it. In practice, one wants to introduce finite delays between successive retransmissions of the same message. Figure 1 presents a slightly more detailed

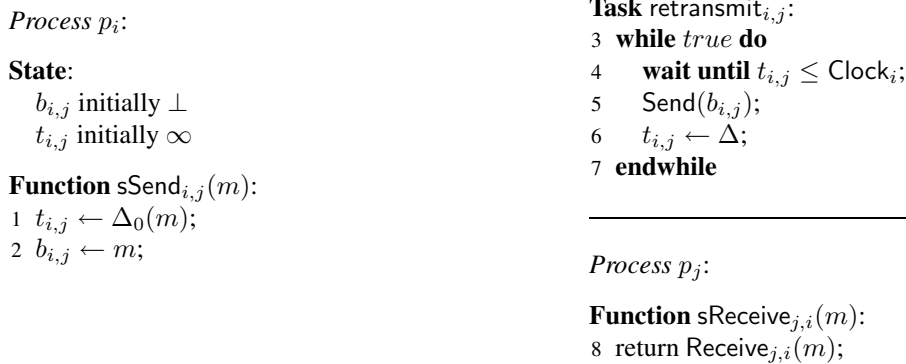


Figure 1: Stubborn channel from p_i to p_j .

version of the implementation of a stubborn channel. Each channel connecting process p_i to process p_j has an associated buffer $b_{i,j}$, initially undefined, a timeout value $t_{i,j}$, initially ∞ , and a background task.

When a message is sent, the timeout is set to Δ_0 (line 1). The message is also stored in the buffer $b_{i,j}$ (line 2). Eventually, as Δ is a finite quantity, the background task wakes up (line 4) and sends the message using the underlying fair-lossy channel (line 5). A new timeout value is then computed (line 6). Message reception translates directly into the underlying message reception primitive (line 8).

Notice that if $sSend(m)$ is executed infinitely often, it is possible that no message is ever transmitted at all, as the timeout is infinitely often updated. On the other hand, if process p_i infinitely delays using $sSend(m)$, then the last message sent which is stored in $b_{i,j}$ will be retransmitted infinitely often thus ensuring that it is eventually delivered.

3.2 Algorithm

In Figure 2 we present an algorithm based on stubborn channels to solve the consensus problem [4]. The algorithm proceeds in asynchronous rounds of two phases. Each round has a designated coordinator that tries to impose its proposal as the decision value. In phase 1, if a majority of the processes endorse the value proposed by the coordinator a decision is locked and processes can decide. However, if the coordinator is suspected to have failed, then processes are requested to enter phase 2 and, as soon as a majority does so, they proceed to the next round. The asynchrony of the rounds means that processes do not need to synchronize when changing rounds and thus we may have different processes in different rounds. Moreover, due to the unreliability of the communication channels, processes are not guaranteed to receive all messages and thus processes may be forced to skip certain rounds.

```

Process  $p_i$ :
Function Consensus( $v_i$ ):
1 ( $est_i.val, est_i.proc$ )  $\leftarrow (v_i, i)$ ;  $r_i \leftarrow 1$ ;
2 while true do
3    $ph_i \leftarrow 1$ ;  $P_i \leftarrow \emptyset$ ;  $coord_i = (r_i \bmod n) + 1$ 
4   if  $i = coord_i$  then
5      $P_i \leftarrow \{i\}$ ;  $est_i.proc = i$ 
6     forall  $k$ :  $sSend_{i,k}((r_i, ph_i, P_i, est_i))$ ;
7   endif;
8   while  $\#P_i \leq n/2$  do
9     select
10    upon  $sReceive_{i,j}((r_j, ph_j, P_j, est_j))$ :
11      if  $r_i < r_j$  then
12         $est_i \leftarrow est_j$ ;  $r_i \leftarrow r_j$ ;
13         $ph_i \leftarrow ph_j$ ;  $P_i \leftarrow \emptyset$ ;
14      endif;
15      if  $r_i = r_j \wedge ph_i < ph_j$  then
16         $ph_i \leftarrow ph_j$ ;  $P_i \leftarrow \emptyset$ ;
17      endif;
18      if  $(r_i = r_j \wedge P_j \setminus P_i \neq \emptyset) \vee$ 
19         $(ph_j = 1 \wedge \#P_j > n/2)$  then
20         $P_i \leftarrow P_i \cup P_j \cup \{i\}$ ;
21      if  $est_j.proc = coord_i$  then
22         $est_i = est_j$ ;
23      endif;
24      forall  $k$ :  $sSend_{i,k}((r_i, ph_i, P_i, est_i))$ ;
25    endif;
26    upon Suspected $_i(j)$ :
27      if  $j = coord_i \wedge ph_i = 1$  then
28         $ph_i \leftarrow 2$ ;  $P_i \leftarrow \{i\}$ ;
29        forall  $k$ :  $sSend_{i,k}((r_i, ph_i, P_i, est_i))$ ;
30      endif;
31    endselect;
32  endwhile;
33  if  $ph_i = 1$  then return  $est_i.val$ ; endif
34   $r_i \leftarrow r_i + 1$ ;
35 endwhile

```

Figure 2: Mutable consensus.

In detail, each process p_i maintains a round (r_i) and a phase (ph_i) counter, an estimate of the decision (est_i), and a set of voters (P_i). The set P_i contains in phase 1 the processes that p_i knows have endorsed the estimate of the current coordinator or, in phase 2, the processes that proceeded to phase 2 and are thus detractors of the coordinator.

In each round the coordinator records its own vote and initiates the round by sending its set of voters and its estimate to all participants (lines 5 and 6).

A round lasts until a majority of votes have been collected (while loop of lines 8 to 32). This set of votes can be either from phase 1 (an endorsement of the coordinator's estimate) and if so a decision is reached (line 33), or from phase 2 and the process proceeds to the next round.

During a round, the handling of a message may undergo two processing steps corresponding to the conditional clauses upon reception (lines 10 to 25). Consider a message m sent by p_j and received by p_i . Firstly, p_i checks whether m comes from a larger round and if so p_i adopts the message's estimate and jumps to the round and phase of m . This is due to the use of stubborn channels as there's no guarantee that p_i receives any messages p_j might have sent to p_i before m and that would enable p_i to proceed. The next clause handles messages from phase 2 of the same round, taking p_i to phase 2 and making it a detractor of the current coordinator.

The second processing step of the message deals with voting. Depending on the phase p_i is in, it may be processing votes supporting the coordinator's proposal (phase 1) or votes to leave the

current round and to proceed (phase 2). Both cases are not distinguished though and are dealt in the same way. When the received message is from the same round p_i is in and brings new votes ($P_j \setminus P_i \neq \emptyset$), then p_i records the new votes adding its own vote (line 20), adopts the message's estimate if it has the coordinator's vote and relays its new set of votes to all processes. This very same processing is done when the received message brings a majority set of votes for phase 1 regardless of the round they were sent. These messages are actually decision messages: p_i records a majority set in P_i , leaves the while loop of line 8, and since it is in phase 1 it returns from function Consensus.

Suspensions are handled in lines 26 to 30. If the suspected process is the coordinator for the current round the process enters phase 2, sending its updated state to all participants. Upon reception of such message, processes still in the first phase of the same round are brought to the second phase (lines 15 to 17).

We assume that the channel receive and failure suspecter primitives in lines 10 and 26 are fair. Therefore, no message is forever pending and not received. Likewise, no suspicion is forever pending and not acknowledged.

3.3 Correctness Argument

Due to lack of space, we omit a correctness proof.¹ Nevertheless, consider the following argument that the algorithm ensures validity and agreement. If a decision on v is reached in some round r , then 1) v is the estimate value est of the coordinator of r , and 2) any process p_i reaching a round $r' > r$ has $est_i = v$. Combining 1) and 2) it is clear that any decision reached in a round $r' > r$ must be on v .

For the first clause it is easy to verify that all messages in phase 1 carry est of the current coordinator and that it is the only value that can be adopted by the other processes as their own est on which they may decide.

With respect to 2), a process p_i can reach $r' > r$ either a) by receiving a message from round r' (lines 11 to 14) or b) by executing line 34 in round r . In order to show 2) we derive a contradiction. Let us consider the first process p_i reaching $r' > r$ with $est_i \neq v$. In case a) p_i reaches r' with the estimate of the process that sends the message from r' (line 12) which would contradict the fact that p_i is the first to reach $r' > r$ with $est_i \neq v$. In b), p_i needed to collect a majority of votes in phase 2 of round r . Since, by assumption, a decision has been reached on v in round r , a majority of processes adopted $est = v$. The intersection of these majorities makes at least one of the messages collected by p_i in phase 2 to contain the estimate v of the coordinator which p_i uses to set est_i in line 22. Process p_i thus leaves round r with $est_i = v$, contradicting the hypothesis

¹For the convenience of reviewers, we include a proof in the Appendix

Process p_i :

Function $\Delta_0(m)$:

```
1 if fresh( $b_{i,j}$ ,  $m$ )  $\vee$  maj( $m$ ) then  
2   return Clock $_i$ ;  
3 else  
4   return Clock $_i$  +  $e$ ;  
5 endif;
```

Function Δ :

```
6 return Clock $_i$  +  $e$ ;
```

Figure 3: Early mutation.

Process p_i :

Function $\Delta_0(m)$:

```
1  $c \leftarrow (\text{round}(m) \bmod n) + 1$   
2 if ( $(i = c \vee j = c) \wedge$  fresh( $b_{i,j}$ ,  $m$ ))  $\vee$  maj( $m$ ) then  
3   return Clock $_i$ ;  
4 else  
5   return Clock $_i$  +  $e$ ;  
6 endif;
```

Function Δ :

```
7 return Clock $_i$  +  $e$ ;
```

Figure 4: Centralized mutation.

and confirming clause 2).

4 Protocol Mutations

In this paper we propose four different implementations of stubborn channels, thus obtaining four protocol mutations. We call them *early*, *centralized*, *ring* and *permutation gossip*. Although we are free to use any implementation, as long as we prove its correctness, we choose to derive all the proposed mutations from Figure 1 just by instantiating functions $\Delta_0(m)$ and Δ . This trivially ensures their correctness. We make the assumption that these functions can read the local state associated with the channel, *i.e.*, $t_{i,j}$ and $b_{i,j}$.

Some of these mutations use the semantics of messages exchanged by the consensus algorithm. In detail, the computation of delays involves evaluation the following conditions of buffered mes-

Process p_i :

Function $\Delta_0(m)$:

```
1 if  $j = ((i + 1) \bmod n) + 1 \wedge (\text{fresh}(b_{i,j}, m) \vee \text{maj}(m))$  then  
2   return  $\text{Clock}_i$ ;  
3 else  
4   return  $\text{Clock}_i + e$ ;  
5 endif;
```

Function Δ :

```
6 return  $\text{Clock}_i + e$ ;
```

Figure 5: Ring mutation.

sages:

$$\begin{aligned}\text{fresh}(b, m) &\equiv b = \perp \vee \text{round}(m) \neq \text{round}(b) \vee \\ &\quad \text{phase}(m) \neq \text{phase}(b) \\ \text{maj}(m) &\equiv \text{voters}(m) > n/2\end{aligned}$$

where for any consensus message $m = (r, ph, P, est)$ we assume $\text{round}(m) = r$, $\text{phase}(m) = ph$ and $\text{voters}(m) = P$.

Early The *early* implementation of Figure 3 is the simplest and aims at a message exchange pattern similar to that of early consensus [5], in which in every round every process multicasts its vote to all others, thus allowing decisions to occur in two communication steps.

Each newly arrived message should be immediately transmitted if it is the first being sent, from a new round or phase or carries a majority of votes (line 1). Such immediate transmission is achieved by returning a value $\leq \text{Clock}_i$ thus allowing the background task to run. If not (line 4), the transmission time is set to the current time plus e . The parameter e (used throughout this section) is an estimate of the time required to finish running an instance of consensus thus attempting that the message will not be actually transmitted. This makes it unlikely that all messages but the first (with the coordinator's and its own vote) and the last (with a majority of votes) are actually sent obtaining the desired result.

Centralized The *centralized* implementation of Figure 4 aims at producing a message exchange pattern which resembles that of the Chandra-Toueg centralized algorithm [4]. In contrast to the *early* mutation, this one does not reproduce exactly the original protocol [4], as the coordinator does not gather estimates when it enters a round. In fact, it differs from the *early* mutation only

<p><i>Process</i> p_i:</p> <p>State: c initially 0 u initially a permutation of $1 \dots n$</p> <p>Function $\Delta_0(m)$: 1 $v \leftarrow \text{turns}_F(u, c, j)$; 2 $c \leftarrow c + v + 1$; 3 return $\text{Clock}_i + ve$;</p>	<p>Function Δ: 4 $v \leftarrow \text{turns}_F(u, c, j)$; 5 $c \leftarrow c + v + 1$; 6 return $\text{Clock}_i + (v + 1)e$;</p> <p>Function $\text{turns}_F(u, c, j)$: 7 $x \leftarrow 0$; 8 while $\nexists l : 0 \leq l < F \wedge$ 9 $u[(l + F(c + x)) \bmod n] = j$ do 10 $x \leftarrow x + 1$; 11 endwhile; 12 return x;</p>
---	--

Figure 6: Permutation gossip mutation.

by avoiding the direct transmission of votes among participants. Instead, these are relayed by the round coordinator.

Therefore, if the sender or the receiver process is the current coordinator, the same delays of the *early* implementation are used. If not, messages are delayed (line 5) by the estimated time required to end consensus e , thus avoiding their transmission. Messages carrying a decision are never delayed.

Ring One can also achieve innovative message exchange patterns with desirable performance characteristics. The *ring* implementation of Figure 5 delays messages from a process i to a process j unless $j = ((i + 1) \bmod n) + 1$. The result is a ring-style message exchange pattern in which each process communicates only with its successor.

Permutation Gossip Gossip-based protocols are used for a variety of distributed programming problems and are known for their scalability and resilience to network omissions. The *permutation gossip* mutation aims at a gossip-style message exchange pattern for consensus with deterministic safety and liveness.

The *permutation gossip* mutation works as follows. Each process generates a random permutation of the sequence of process identifiers. This sequence is used as a circular list. Upon sending a message, it is transmitted immediately to the next F processes (fanout value) in such list and delayed for all other processes. The pointer into the list is then incremented by F . In contrast with other mutations, not all transmissions are equally delayed. In fact, each transmission is delayed such that after every period e it occurs for the next F processes in the list. Eventually, in at most n/F periods, the message will have been transmitted to all processes. This repeats every n/F periods.

The algorithm for the *permutation gossip* is presented in Figure 6. Function $\text{turns}_F(u, c, j)$ computes the number of turns until the next transmission of the message for destination p_j . This is done by incrementing counter x (line 9) until j is within the next F identifiers in the list. The initial transmission is then scheduled accordingly (line 3). Retransmissions are scheduled with identical procedure (lines 4 to 6).

5 Evaluation

We use a pragmatical approach to evaluate the mutable consensus protocol: an implementation is tested in a realistic environment which accounts for CPU and network overhead. This approach allows us to balance communication steps with the number of messages transmitted and handled by a single process. In this section we describe the implementation, the testing environment and then present the results.

5.1 Implementation

The implementation of the mutable consensus protocol and of stubborn channels used for evaluation is based on Java. A single thread is used for each process, as the code is structured as a set of event handlers. The main loop executed by the thread is in charge of setting timers and polling a datagram socket for incoming messages, invoking then the appropriate event handlers.

Round and phase numbers are represented by 32 bit integers. The set of voters P_i is represented as a bitmap and stored as an array of 32 bit integers, making it compact for transmission and reducing set union to a bitwise or operation. Conversion between internal and external representation is done using standard classes (*java.io.DataOutputStream* and *java.io.DataInputStream*). A custom buffer class is used for handling messages, which are handled internally as a list of byte arrays. Efficient methods for adding and removing headers are provided. Values proposed for consensus are also represented as message buffers.

5.2 Simulation Runtime

Common metrics often misrepresent the performance of distributed algorithms in complex environments such as the Internet [11]. Therefore we use a centralized simulation model [12] to evaluate the performance of the protocol. Centralized simulation works as follows: the execution of real code is timed with a high resolution clock and the resulting elapsed time is used to update simulated time-lines associated with simulated processors in the context of a discrete event simulation model. Such models have been shown to accurately reproduce the timing characteristics of real systems [12] and have several advantages: only a single host is required, even when testing

configurations with a large number of processes and arbitrarily complex networks; it is possible to perform global observations, including time durations; and it is very easy to perform fault injection to test and evaluate distributed fault-tolerant programs.

Such models have been shown to accurately reproduce the timing characteristics of real systems [12] and have several advantages when compared to testing in real systems. First, only a single host is required, even when testing configurations with a large number of processes and arbitrarily complex networks. It is also possible to perform global observations, including time durations and to perform fault injection to test and evaluate distributed fault-tolerant programs.

The centralized simulation runtime used is also based on Java and uses a virtual per-process CPU cycle counter to measure time [13]. By comparing the results of simple benchmarks run in the simulated environment and in the corresponding real system, one can derive which configuration parameters to use in simulated components and ensure that the results closely match [12].

5.3 Simulated Environment

The configuration used to obtain the results presented in this paper was tuned to reproduce Pentium III 1GHz workstations running Linux 2.4.21 and Sun HotSpot JVM 1.4.2. The model used does not however simulate scheduling latency, thus providing results that can only be observed in a real system if no other tasks are competing for the CPU or if a sufficiently higher priority is assigned to the protocol task.

The simulated network mimics a switched 10Mbits Ethernet (*i.e.*, star topology). The model includes the transmission delay as well as event scheduling and operating system overhead. Buffers are also calibrated according to a real system in order to accurately reproduce message loss when the system is congested. Nevertheless, the bandwidth can be varied and arbitrary packets dropped to stress protocols. The failure detector is also simulated and can be configured to generate specific patterns. Processes can also be crashed in specific runs of the protocol. Unless noted otherwise, results presented in this paper are obtained without process crashes or suspicions.

The simulated application works as follows: Values are proposed simultaneously by all processes. Each value is an empty message buffer, thus ensuring that the results obtained are entirely due to protocol overhead. When all processes decide they are restarted thus initiating a new run of consensus. The network is not restarted, although stale packets are dropped upon reception.

Performance results are obtained in two steps. First, significant events are logged to files while the simulation is running. When the simulation has stopped, log files are processed to extract relevant statistics.

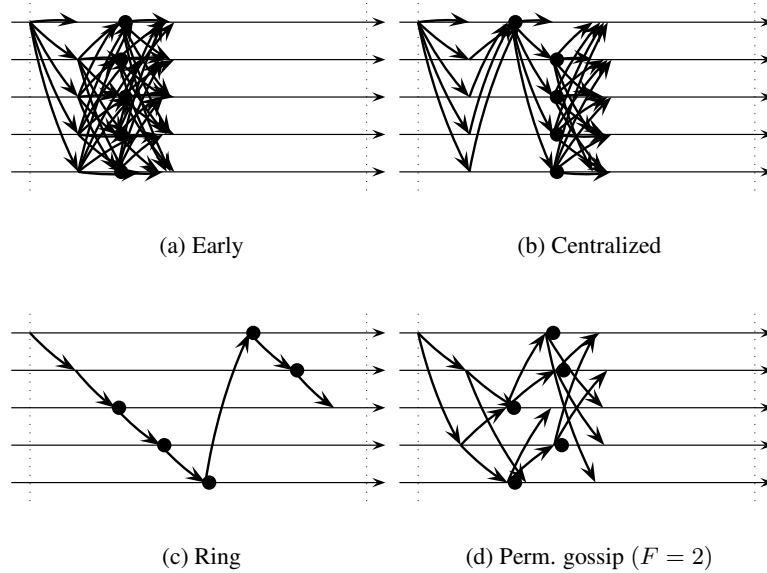


Figure 7: Prefixes of typical executions (1ms).

5.4 Results

A first intuition on the impact of such delays can be obtained from Figure 7, which presents the graphical representation of prefixes of actual runs of the mutable consensus protocol when combined with each implementation of channels. In these pictures, arrows denote messages and solid dots the return from the Consensus function (*i.e.*, the decision). The x -axis represents real-time. The entire duration of the interval presented is 1 ms. All messages actually transmitted during the 1ms interval are presented. All mutations are configured with $e = 2ms$ and *permutation gossip* with $F = 2$.

Although individual runs presented in Figure 7 provide an intuition on the behavior of the protocol, the overall performance is better evaluated by statistics on protocol latency and resource usage. Figure 8(a) shows the latency, from proposal to decision, of the consensus function as seen by one process other than the coordinator. Notice that with a small number of processes, the *early* mutation offers the best results. As expected, the latency of the *ring* mutation grows linearly with the number of processes. The latency of the *permutation gossip* mutation grows logarithmically.

The latency of protocol mutations closely follows the average number of messages processed presented in Figure 9(b). The exception is the *ring* mutation, in which the latency is justified by the low resource consumption.

The sudden increase of latency of *early* and *centralized* mutations is explained by Figure 8(c), which shows the average bandwidth consumed. It turns out that the corresponding network link

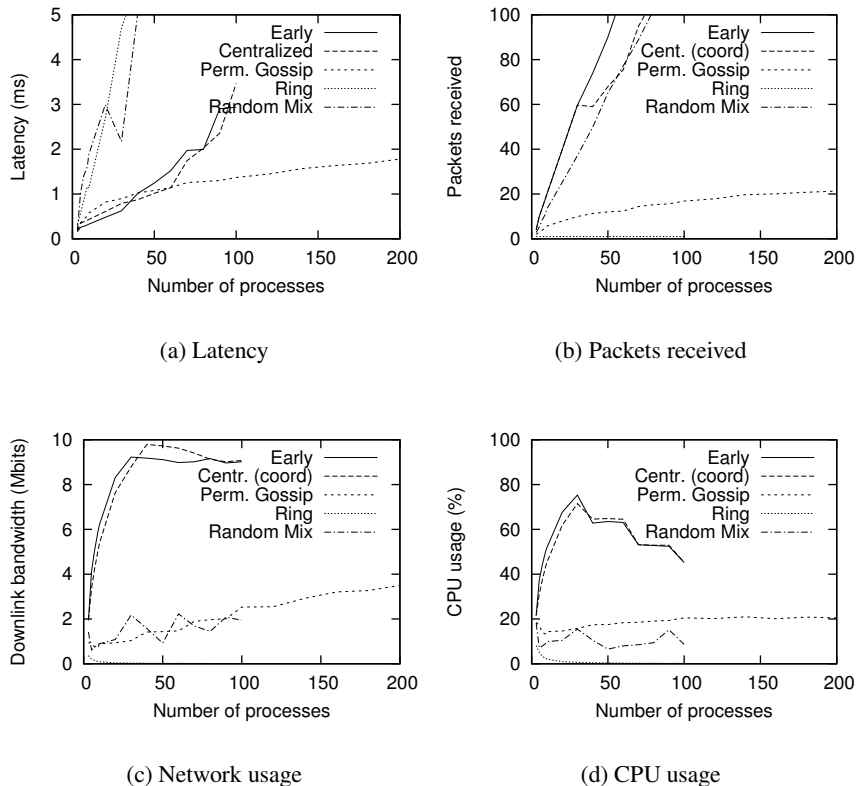


Figure 8: Performance of protocol mutations.

in the switched Ethernet becomes saturated leading to messages being discarded, retransmissions and a longer time to complete. In contrast, network usage is extremely low with the *ring* mutation and moderate with the *permutation gossip*, even with a very large number of processes.

The effect of network congestion is also visible in Figure 8(d), which displays average CPU usage. Notice that with a small number of processes, the *early* mutation makes the most efficient usage of resource, therefore justifying the better latency. Nevertheless, when the network is congested it becomes the bottleneck and thus the CPU becomes idle. This is bad, as the system is doing nothing else than solving consensus. In contrast, the *ring* mutation makes a very poor usage of CPU, as processes are most of the time idle waiting for messages. In between, the *permutation gossip* mutation allows a fair usage of CPU thus justifying its performance.

One concludes that both the *early* and *centralized* mutations do not scale regarding network and CPU usage. The *early* mutation is however still the best choice for small groups (*e.g.* less than 20 processes). Interestingly, almost all protocols proposed so far [4, 5, 14, 9, 15] rely on a similar message exchange pattern in which at least one process receives messages from all others thus sharing the same scalability limitations.

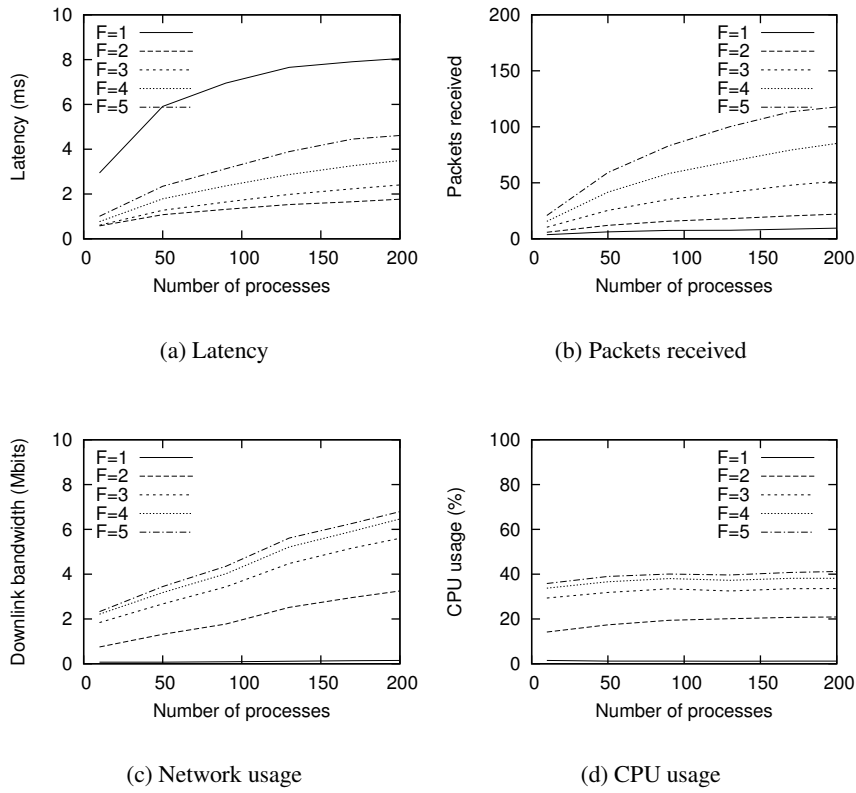
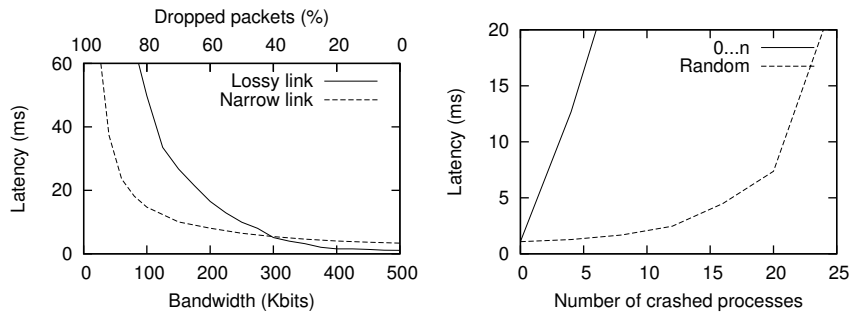


Figure 9: Protocol parameters.

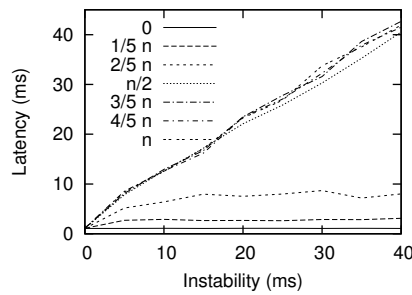
The *ring* mutation is extremely frugal in terms of resource consumption, although resulting in high latency. It would however be desirable if a large number of consensus instances would be running simultaneously and latency is not a primary concern. Finally, the *permutation gossip* is scalable to a large number of processes while at the same time offering low latency. Such message exchange pattern is also highly resilient to process failure and network omissions.

Figure 8 includes also results labeled as *random mix*. These were obtained by making each process randomly select with equal probabilities which mutation to use for each consensus instance. Although the performance is not good by any metric, the fact that processes do not block confirms the correctness of the approach. The poor performance is explained by the fact that processes using the *centralized* or *ring* mutations communicate only with a single other process until period e expires, without the guarantee that such process does the right thing performance-wise.



(a) Variable bandwidth

(b) Crashed processes



(c) False suspicions

Figure 10: Environment conditions.

6 Configuration of Permutation Gossip

It is clear from the previous section that the *permutation gossip* mutation is the most interesting, in particular, with a large number of participants. In this section we examine in detail the impact of configuration parameters F and e as well as of environment conditions.

6.1 Protocol Parameters

The performance of the *permutation gossip* mutation is intimately related with the fanout parameter F . Intuitively, higher values of F should allow faster dissemination of votes, at the expense of higher resource consumption. Figure 9 shows the effect in performance and resource consumption of using different values for F . From the results it is clear that the best results are obtained with $F = 2$. This is justified by every reception of a message containing new votes causing a new message to be sent. It is therefore useless to flood the system with lots of messages carrying the same set of votes.

We have also experimented different values for gossip period e and drawn the conclusion that,

with $F = 2$ and e larger than the expected duration of a full run, the performance of the protocol is independent of e .

6.2 Environment Conditions

Gossip-based protocols are known for their resilience to network omission failures and process crashes. By using the centralized simulation facility we can easily crash processes, inject false suspicions, random loss and cause congestion to evaluate the protocol. We use an initial set of 50 processes for each of the experiments.

Figure 10(a) shows the effect of forcing the network to drop a variable amount of messages (solid line, top axis). It is clear that the protocol is still providing a very good performance with up to 40% of packets discarded. The protocol collapses only when more than 80% of messages are lost.

Dropping messages uniformly and regardless of load is not however a realistic model of how networks behave. Real loss is better modeled by constraining the available bandwidth (dashed line, bottom axis). Although the protocol is normally using more than 1Mbits (see Figure 9(c)), the protocol is still functioning reasonably well with as little as 250Kbits and collapses only with less than 50Kbits.

Figure 10(b) shows the impact of crashed processes in the performance of the protocol. The worst case scenario for a protocol that deterministically selects a coordinator is when the first processes to be selected are all crashed. Therefore, when k processes have crashed the protocol is forced to go through $k + 1$ rounds, resulting in a linear increase in latency as shown by the solid line, even when the failure detector is behaving perfectly. The more likely scenario is that a random set of processes has crashed. This results in much better performance as gossiping is able to route around crashed processes. Performance degrades seriously only when the number of failures approaches $n/2$.

False suspicions have a similar impact in protocol performance by forcing the protocol to skip several rounds until a decision is reached. It is not as serious as all processes are still able to relay messages. Figure 10(c) shows the impact of a variable share of processes (selected at random) having their failure detector oracles falsely suspect all coordinators for a given period of time. It can be observed that when the number of processes with misbehaving detectors is greater than $n/2$, latency is directly proportional to the instability period. When the number of processes is equal to or less than $n/2$, there is some impact but the protocol quickly converges to a decision despite the instability.

7 Related Work

Although we focus on consensus protocols based on unreliable failure detectors, a number of other additions to an asynchronous system model has been proposed in order to make consensus solvable (*e.g.* [16, 17]). It should be possible to obtain performance advantages with mutable protocols on different system models, as long as these rely on gathering votes to reach decisions.

The proposal of generalized consensus protocols has been done before, in particular regarding also the communication pattern [9] and the oracle used [15]. The first approach [9] also addresses the trade-off between latency and bandwidth, but is less flexible in terms of what communication patterns can be obtained. Specifically, it cannot be instantiated to mimic the ring or the gossip mutations introduced here and requires the coordination of processes on the pattern used. The second approach [15] addresses only the issue of which oracle to use. This is orthogonal to our proposal and it should be possible to combine them.

Gossip-based protocols have been used in several protocols to solve other distributed programming problems. Namely, for probabilistic reliable multicast[18] and stability detection [19]. In common, such protocols offer good scalability to a large number of participants and resiliency to process and network omission failures.

In contrast with protocol configuration by layer switching [20], no coordination at all is required when selecting the strategy used to compute delays in mutable consensus. In fact, different processes may be simultaneously using different strategies without endangering correctness. This means also that, given an adequate policy, it is trivial do dynamically reconfigure the protocol to adapt to a changing environment.

8 Conclusions

The mutable consensus protocol is interesting from a theoretical point of view, as it abstracts the behavior of several (apparently) distinct consensus protocols. Furthermore, the performance tuning procedure operates only in the time domain and thus does not, in any way, affect the correctness of protocol which assumes an asynchronous system model. This has interesting consequences. First, strategies used in the computation of delays required for mutations can be arbitrarily complex and varied without requiring additional correctness proofs. As an example, in addition to using the semantics of messages as shown in this paper, it is also interesting to research using information from the environment (*e.g.* network conditions). Finally, no coordination at all is required when selecting the strategy used to compute delays. In fact, different processes may be simultaneously using different strategies without endangering correctness. Given an adequate policy, it is trivial do dynamically reconfigure the protocol to adapt to a changing environment.

Although the mutations explored are very simple, the performance obtained by the *permutation gossip* introduced in this paper is already very good. In fact, in the realistic setting used for evaluation in this paper it surpasses other consensus algorithms in scalability to large numbers of processes. It is also clear that unless the number of processes is small and there are plenty of resources, the amount of messages to be handled by a single process is more relevant for performance than the number of communication steps required to decide. This underlines the usefulness of mutating the protocol depending on the system configuration.

Notice also that protocol mutation is possible because: (i) the information received is always relayed and (ii) the protocol assumes lossy channels. The second is particularly interesting, as it precludes obtaining similar performance advantages from higher level mutable protocols based on reliable multicast. To make it possible, one should use semantically reliable multicast, which generalizes the stubborn channel abstraction to multicast communication [21]. One can also consider developing mutable protocols for distributed programming problems other than consensus. In fact, the implementation of mutable consensus presented here is part of GROUPZ, a group communication toolkit based on mutable protocols which is configured by selecting implementations of stubborn channels.

References

- [1] J. Pereira and R. Oliveira, “A mutable protocol for consensus in large groups,” in *Ws. on Large Scale Group Communication (with SRDS’2003)*, 2003.
- [2] R. Guerraoui and A. Schiper, “The generic consensus service,” *IEEE Transactions on Software Engineering*, vol. 27, no. 1, Jan. 2001.
- [3] M. Fischer and N. Lynch and M. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM*, Apr. 1985.
- [4] T. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM*, vol. 43, no. 2, Mar. 1996.
- [5] A. Schiper, “Early consensus in an asynchronous system with a weak failure detector,” *Distributed Computing*, vol. 10, no. 3, Apr. 1997.
- [6] O. Bakr and I. Keidar, “Evaluating the running time of a communication round over the internet,” in *ACM Symp. Principles of Distributed Computing*, 2002.

- [7] R. Guerraoui and R. Oliveira and A. Schiper, “Stubborn communication channels,” Tech. Rep.98-278, Département d’Informatique, École Polytechnique Fédérale de Lausanne, June 1998.
- [8] A. Basu and B. Charron-Bost and S. Toueg, “Simulating reliable links with unreliable links in the presence of process crashes,” in *International Workshop on Distributed Algorithms*, Oct. 1996.
- [9] M. Hurfin and A. Mostefaoui and M. Raynal, “A versatile family of consensus protocols based on Chandra-Toueg’s unreliable failure detectors,” *IEEE Transactions on Computers*, vol. 51, no. 4, Apr. 2002.
- [10] R. Guerraoui, “Indulgent algorithms,” in *ACM Symposium on Principles of Distributed Computing (PODC’00)*, July 2000.
- [11] I. Keidar, “Challenges in evaluating distributed algorithms,” vol., 2584 of *Lecture Notes in Computer Science*. Springer, 2003.
- [12] G. Alvarez and F. Cristian, “Simulation-based testing of communication protocols for dependable embedded systems,” *The Journal of Supercomputing*, vol. 16, no. 1, May 2000.
- [13] M. Pettersson, “Linux performace monitoring counters,” <http://www.csd.uu.se/~mikpe/linux/perfctr/>.
- [14] R. Oliveira, *Solving Consensus: From Fair-Lossy Channels to Crash-Recovery of Processes*, Ph.D. thesis, Département d’Informatique, École Polytechnique Fédérale de Lausanne, Feb. 2000.
- [15] R. Guerraoui and M. Raynal, “The information structure of indulgent consensus,” Tech. Rep.PI-1531, IRISA, Apr. 2003.
- [16] M. Ben-Or, “Another advantage of free choice: Completely asynchronous agreement protocols,” in *ACM Symp. on Principles of Distributed Computing (PODC’83)*, 1983.
- [17] T. Chandra and V. Hadzilacos and S. Toueg, “The weakest failure detector for solving consensus,” *Journal of the ACM*, vol. 43, no. 4, July 1996.
- [18] K. Birman and M. Hayden and O. Ozkasap and Z. Xiao and M. Budiu and Y. Minsky, “Bi-modal multicast,” *ACM Transactions on Computer Systems*, vol. 17, no. 2, 1999.

- [19] K. Guo and M. Hayden and R. van Renesse and W. Vogels and K. Birman, “GSGC: An efficient gossip-style garbage collection scheme for scalable reliable multicast,” Tech. Rep. TR97-1656, Cornell University, Computer Science Department, Dec. 1997.
- [20] X. Liu and R. van Renesse and M. Bickford and C. Kreitz and R. Constable, “Protocol switching: Exploiting meta-properties,” in *IEEE Intl. Ws. Applied Reliable Group Communication*, 2001.
- [21] Pereira, J. and Rodrigues, L. and Oliveira, R., “Semantically reliable multicast: Definition, implementation and performance evaluation,” *IEEE Transactions on Computers*, vol. 52, no. 2, Feb. 2003.

A Correctness of the algorithm

We prove the correctness of the algorithm of Figure 2 assuming that a majority of the processes are correct and that the failure detector is of class $\diamond S$ [4]. We say that a process p_i *reaches round* ρ when the current round r_i of p_i is such that $r_i \geq \rho$, and we say that p_i *reaches phase 2 of round* ρ if either (1) $r_i = \rho$ and $ph_i = 2$, or (2) $r_i > \rho$. A process p_i *enters round* ρ when p_i sets $r_i = \rho$, and *enters phase 2 of round* ρ when $r_i = \rho$ and p_i sets $ph_i = 2$.

Lemma A.1 (Termination-1) *If one correct process decides, then every correct process eventually decides.*

PROOF: Let p_i be a correct process that decides (line 33). To do so p_i must have collected a majority of votes of phase 1 and have sent message (r_i, ph_i, P_i, est_i) to all processes as its last message (line 24). By the Stubborn property of the communication channels all correct processes eventually receive the message. This message carries a majority of voters in P_i enabling its receiver, no matter in which round it is (line 19), to adopt the majority set of voters, est_i and decide.

Lemma A.2 (Termination-2) *Let ρ be any round such that $\rho > 0$. The first process to reach round ρ leaves round $\rho - 1$ at line 34.*

PROOF: Assume that p_i is the first process to reach round ρ . To reach round ρ , either (1) p_i executes line 34 of round $\rho - 1$, or (2) in a round smaller than ρ process p_i receives at line 10 a message (r_j, ph_j, P_j, est_j) from some process p_j in round $\rho' \geq \rho$ making p_i to set r_i to r_j . Case (2) is in contradiction with the assumption that p_i is the first process to reach round ρ , and so only case (1) is possible.

Lemma A.3 (Termination-3) *Let ρ be any round. The first process to reach phase 2 of round ρ suspects the coordinator of round ρ .*

PROOF: A process p_j reaches phase 2 of round ρ by setting either (1) $r_j = \rho$ and $ph_j = 2$, or (2) $r_j = \rho' > \rho$. In case (2), by Lemma A.2, some process p_i (possibly $p_i = p_j$) must have already left round ρ at line 34. To leave round ρ at line 34, p_i must have set $r_i = \rho$ and $ph_i = 2$ (case (1)). Therefore, the first process p_i to reach phase 2 of round ρ does so with $r_i = \rho$ and $ph_i = 2$.

Consider p_i as the first process to reach phase 2. Process p_i can set $ph_i = 2$, either (1) at line 28, (2) at line 12 or (3) at line 16. However, to execute lines 12 or 16 (cases (2) and (3)), p_i

must have received a message from some process p_j that already set $ph_j = 2$ in round ρ , which contradicts the assumption that p_i is the first process to do so. Therefore, only case (1) is possible and, to execute line 28, p_i must suspect the coordinator of round ρ at line 26.

Lemma A.4 (Termination-4) *There is a round ρ^* such that no process reaches any round larger than ρ^* .*

PROOF: (1) Let t_c be the time by which all incorrect processes have already crashed. (2) By the *eventual weak accuracy* property of $\diamond S$, there is a time t_a and a correct process p_c such that no correct process suspects p_c after t_a . Let t be the largest of times t_c and t_a . Clearly, by (1) and (2), after t no running process suspects p_c .

Let r_t be the largest round that some process has reached at time t , and $\rho^* > r_t$ the smallest round in which p_c is the coordinator. The proof of the lemma is by induction on the round number.

Base step: No process reaches round $r = \rho^* + 1$.

The proof is by contradiction. Let p_i be the first process to reach round $\rho^* + 1$. By Lemma A.2, process p_i can only proceed to round $\rho^* + 1$ if p_i executes line 34 at round ρ^* . To do so, p_i must have set $ph_i = 2$ either (1) at line 28 or (2) at lines 12 or 16. This implies that (1) p_i suspects p_c , or (2) p_i receives a message from some process p_j in phase 2 of ρ^* . From the processes that reached phase 2 of ρ^* , consider the first that did so. By Lemma A.3, this process suspected p_c in round ρ^* .

Both cases are impossible since round ρ^* happens after t and no process executing round ρ^* suspects p_c after time t . A contradiction.

Induction step: If no process reaches a round $r = \rho^* + 1$, then no process reaches round $r + 1$.

By Lemma A.2, the first process to reach round $r + 1$ leaves round r at line 34. By the induction hypothesis, no process reaches round r , and thus no process reaches round $r + 1$.

Lemma A.5 (Termination-5) *For any round ρ , if no correct process ever decides, then every correct process eventually reaches round $\rho + 1$.*

PROOF: Let ρ be the smallest round for which the Lemma does not hold, *i.e.*, no correct process ever decides, and some correct process never reaches round $\rho + 1$. As ρ is the smallest of such rounds, each correct process eventually reaches round ρ . We will contradict the hypothesis by proving the following successive results:

- i) At least one correct process eventually reaches phase 2 of round ρ .

- ii) Each correct process eventually reaches phase 2 of round ρ .
- iii) Each correct process eventually reaches round $\rho + 1$.

If a correct process sends a message m as its last message, then by the the Stubborn property of the communications channels all correct processes eventually receive m . To avoid cluttering the text, we will use this fact implicitly throughout the proof.

Proof of (i): Assume for a contradiction that no correct process reaches phase 2 of round ρ . We consider two cases: the coordinator of round ρ , process p_c , (1) is a correct process, or (2) is an incorrect process.

Case (1). Since, by hypothesis, all correct processes reach round ρ and no process reaches phase 2, all processes in round ρ may only receive messages from phase 1 of ρ and process them through lines 18 to 25.

A correct process p_i sends a message at line 24 only if it received a message with (a) new voters ($P_j \setminus P_i \neq \emptyset$), or (b) a majority set of voters from phase 1. Case (b) would lead p_i to decide contradicting the hypothesis that no correct process ever decides. We are left with case (a) and, since the number of processes is finite, processes eventually stop sending new messages at line 24.

Process p_c sends (r_c, ph_c, P_c, est_c) to all processes at line 6. This message is eventually received by each correct process p_i . Process p_i adds itself as a voter and relays the message. Any subsequent message of p_i in phase 1 of round ρ contains a larger set of voters. Eventually, some correct process receives messages from a majority of processes, collects a majority set of voters and decides in round ρ : a contradiction with the fact that no correct process decides.

Case (2). Process p_c is incorrect. By the *strong completeness* property of $\diamond S$, p_c is eventually suspected by every correct process. Therefore, there is at least one correct process that suspects p_c and reaches phase 2 at line 28: a contradiction with the fact that no process reaches phase 2 of round ρ .

Proof of (ii): At least one correct process eventually reaches phase 2 of round ρ . Let p_i be the first correct process to reach phase 2. Process p_i sends a phase 2 message at line 29 and this is the last message p_i may send in round ρ .

Assume for contradiction that there is some correct process p_j that does not reach phase 2. We consider two cases: the last message p_i sends is (1) from phase 2 of round ρ , or (2) from some round $\rho' > \rho$.

In both cases, p_j eventually receives the message of p_i and (1) at line 16 reaches phase 2 of ρ , or (2) at line 12 reaches round ρ' . A contradiction to the fact that there is some correct process that does not reach phase 2 of round ρ .

Proof of (iii): All correct processes reach phase 2 of round ρ . In case (2) of (ii), all correct processes eventually reach some round $\rho' > \rho$ and so reach round $\rho + 1$. Therefore we need only to prove that all correct processes leave round ρ when a message from phase 2 of ρ is the last estimate message any correct process ever sends.

Every correct process p_i reach phase 2 of round ρ . Process p_i sends a phase 2 message at line 29 and this is the last message p_i may send in round ρ .

As every correct process p_i sends a phase 2 message as its last message, all correct processes eventually receive a majority of these messages at line 10 and build up a majority set of voters at line 20. Then, the while loop condition of line 8 becomes false for all correct processes and each correct process proceeds to round $\rho + 1$ at line 34.

Lemma A.6 (Validity) *All messages $(r_i, ph_i, P_i, est2_i)$ sent during phase 1 of round ρ carry the estimate_c value of the coordinator p_c of round ρ .*

PROOF: The proof is by induction on the length of the *send-receive* chain of messages $(\rho, 1, P_i, est_i)$. The messages $(\rho, 1, P_i, est_i)$ are numbered as follows:

- the message $(\rho, 1, P_i, est_i)$ sent by the coordinator at line 6 is numbered 0;
- if the message $(\rho, 1, P_i, est_i)$ received by p_i at line 10 is numbered k , then the message $(\rho, 1, P_i, est_i)$ sent by p_i at line 24 is numbered $k + 1$.

Base step. Trivially, for the message $(\rho, 1, P_i, est_i)$ number 0, est_i is the estimate of the coordinator of round ρ .

Induction step. Consider a message $(\rho, 1, P_i, est_i)$ with number $k + 1$. This message is sent by some process p_i at line 24, after having received, at line 10, the message $(\rho, 1, P_i, est_i)$ with number k . By induction hypothesis, this message carries the est_c value of the coordinator of round ρ . Therefore, because of line 22, the message $(\rho, 1, P_i, est_i)$ also carries the est_c value of the coordinator of round ρ .

Lemma A.7 (Agreement-1) *If a process (correct or incorrect) decides v in round ρ , then v is the estimate_c of the coordinator p_c of round ρ .*

PROOF: A process may only decide on est_i received in a phase 1 message at line 10. By Lemma A.6, the value received is the est_c of the coordinator p_c of round ρ .

Lemma A.8 (Agreement-2) *Let p_c be the coordinator of round ρ . If a process p_i (correct or incorrect) leaves ρ at line 34 with $est_i \neq est_c$, then no process decides in round ρ .*

PROOF: We first consider the conditions for a process (1) to decide in round ρ , and (2) to leave round ρ at line 34.

(1) Let p_k be a process that decides in round ρ at line 33. To decide, p_k must receive phase 1 messages from a majority set of processes at line 10. Let us call this set $ChampionSet_{\rho,k}$: we have $|ChampionSet_{\rho,k}| > n/2$. Let p_j be a any process in $ChampionSet_{\rho,k}$. If $p_j = p_c$ then trivially $est_j = est_c$. Otherwise, by Lemma A.6 and lines 21 and 22 est_j also equals est_c .

(2) Let p_i be a process that leaves round ρ at line 34 with $est_i \neq est_c$. Process p_i must have received phase 2 messages from a majority set of processes at line 10. Let us call this set $DetractorSet_{\rho,i}$: we have $|DetractorSet_{\rho,i}| > n/2$.

We prove that if process p_i leaves round ρ at line 34 with $est_i \neq est_c$ ($|DetractorSet_{\rho,i}| > n/2$), then no process p_k can decide in round ρ ($|ChampionSet_{\rho,k}| \leq n/2$).

(a) The fact that p_i leaves ρ at line 34 with $est_i \neq est_c$ means that no process $p_j \in DetractorSet_{\rho,i}$ has $est_j = est_c$, which implies that p_j did not execute line 24 in phase 1 of round ρ and therefore did not send any phase 1 message.

(b) Because every process $p_j \in DetractorSet_{\rho,i}$ has set $r_j = \rho$ and $ph_j = 2$ before sending phase 2 messages, p_j can no longer vote for a decision in round ρ .

From cases (a) and (b), no process $p_j \in DetractorSet_{\rho,i}$ can also belong to $ChampionSet_{\rho,k}$, hence $DetractorSet_{\rho,i} \cap ChampionSet_{\rho,k} = \emptyset$. By assumption $|DetractorSet_{\rho,i}| > n/2$. Therefore $|ChampionSet_{\rho,k}| \leq n/2$, and thus no process p_k can decide in round ρ .

Lemma A.9 (Agreement-3) *If every process p_i that leaves round ρ at line 34 does so with $est_i = v$, then every process p_j that enters round $\rho' > \rho$ does so with $est_j = v$.*

PROOF: The proof is by induction on the difference between ρ' and ρ . First we prove the lemma for $\rho' = \rho + 1$.

To prove the general case we show that if every process p_i that enters round $\rho' > \rho + 1$ does so with $est_i = v$, then every process p_j that enters round $\rho' + 1$ does so with $est_j = v$.

Base step. $\rho' = \rho + 1$. We have two cases to consider: (1) p_i enters round ρ' leaving round ρ at line 34, (2) p_i enters round ρ' leaving some round $r \leq \rho$ at line 12.

Case (1). By hypothesis, any process p_i that leaves round ρ at line 34 does so with $est_i = v$.

Case (2). The proof is by contradiction: let p_i be the first process to enter round ρ leaving round $r \leq \rho$ at line 12 with $est_i \neq v$. Process p_i leaves round $r \leq \rho$ at line 12 after receiving a message from some process p_j in round ρ' . Process p_i enters round ρ' with $est_i = est_j$ received in

$(\rho', ph_j, P_j, est_j)$ (line 10) which implies that p_j , in round ρ' has $est_j \neq v$. Since, by hypothesis, p_i is the first process to enter round ρ' with $est_i \neq v$, p_j must have changed its $est_j = v$ to $est_j \neq v$ in round ρ' .

Since any process in round ρ' can only change its estimate value at line 22 to the estimate value of some other process already in round ρ' , admitting that p_j changes its estimate to $est_j \neq v$ contradicts the assumption that p_i is the first process to reach round ρ' with $estimate_i.value \neq v$.

Induction step. Assuming that the lemma is verified for $\rho' > \rho$ we shall prove that it is also verified for $\rho' + 1$. Consider a process p_i that enters round $\rho' + 1$. We have two cases: (1) p_i leaves round ρ' at line 34, (2) p_i leaves round $r \leq \rho'$ at line 12.

Case (1). Suppose by contradiction that p_i leaves round ρ' at line 34 with $est_i \neq v$. Since p_i entered round ρ' with $est_i = v$, p_i changed its estimate value in round ρ' . Process p_i can only change its estimate at line 22 with the estimate of some other process in round ρ' .

Let p_k be the first process to change its estimate to $est_k \neq v$ in round ρ' . Because by the induction hypothesis all processes that enter round ρ' do so with $est = v$, and by assumption p_k is the first process to change its estimate, there is no $est \neq v$ process p_k can adopt. Therefore process p_i cannot change its estimate to $est_i \neq v$. A contradiction to the fact that p_i leaves round ρ' with $est_i \neq v$.

Case (2). Similar to the proof of case (2) in the base step.

We now prove, based on the previous lemmas, that the consensus algorithm of Figure 2 satisfies the Termination, Agreement, and Validity properties of Consensus (Section 2.2).

Proposition A.10 (Termination) *The consensus algorithm of Figure 2 satisfies the Termination property.*

PROOF: Let t_c be the time by which all incorrect processes have already crashed. By the *eventual weak accuracy* property of $\diamond S$, there is a time t_a and a correct process p_c such that no correct process suspects p_c after t_a . Let t be the largest of t_c and t_a . Clearly, after t no process suspects p_c .

Let ρ be a round such that (i) p_c is the coordinator of ρ , and (ii) every correct process enters round ρ after t (if such a round does not exist, then by Lemma A.5 one correct process has decided in a round $\rho' < \rho$, and so, by Lemma A.1, every correct process decides, and the Termination property holds). Since no process suspects p_c , by Lemma A.3, no process reaches phase 2 of ρ .

In round ρ , p_c includes itself as a voter in P_c and sends (ρ, ph_c, P_c, est_c) to all processes (line 6). Since no process leaves phase 1, processes exchange phase 1 messages until no new voters are received or the set of voters at each process contain a majority of processes. Because

the number of processes is finite, so it is the number of messages each process may send in phase 1. By the Stubborn property of the communication channels, the last message of each process is guaranteed to be received by all other (correct) processes. Since, by assumption, a majority of processes are correct, eventually some process p_i will gather a majority of voters in P_i , leaves the while loop of line 8 and decides (33). Once p_i has decided, by Lemma A.1 every correct process eventually decides.

Proposition A.11 (Agreement) *The consensus algorithm of Figure 2 satisfies the Agreement property.*

PROOF: Assume that a process (correct or incorrect) decides on v in round ρ . We prove that no other process can decide on a different value.

Consider a decision in round ρ' . If $\rho' = \rho$, by Lemma A.7, any process which decides does so on the est_c value of the coordinator p_c of round ρ , *i.e.*, on the same value. If $\rho' > \rho$ (if $\rho' < \rho$ then assume v as the decision value in round ρ' and rename ρ to ρ' , and ρ' to ρ), by Lemma A.7 v is the est_c value of the coordinator p_c of round ρ , and by Lemma A.8 every process p_i which leaves round ρ at line 34 does so with $est_j = v$. By Lemma A.9 every process p_i which enters round ρ' has $est_i = v$ and so does the coordinator $p_{c'}$ of round ρ' . By Lemma A.7 any process which decides in round ρ' does so on the $est_{c'}$ value of the coordinator $p_{c'}$ of round ρ' , *i.e.*, on v .

Proposition A.12 (Validity) *The consensus algorithm of Figure 2 satisfies the Validity property.*

PROOF: Assume, by contradiction, that Validity does not hold. Then some process (correct or incorrect) p_i sets est_i to a value that is not the proposal of any process. Let ρ be the smallest round in which this happens.

Case (1). Assume that this happens in phase 1 of round ρ , *i.e.*, at line 22. By Lemma A.6 every estimate sent in phase 1 of round ρ is the estimate of the coordinator p_c of round ρ . If $\rho = 0$, est_c is the proposal of p_c . A contradiction. If $\rho > 1$, then est_c is the estimate of p_c at the end of round $r < \rho$. As by hypothesis ρ is the smallest round in which some process p_i sets est_i to a value that is not the proposal of some process, the value est_c is the proposal of some process, and we conclude by contradiction.

Case (2). Assume that this happens in phase 2 of round ρ : some process sets for the first time est_i to a value that is not the proposal of some process in phase 2 of round ρ . This can only occur at line 22, where est_j is the estimate received at line 10. Any estimate received at line 10 is sent by some process either at line 24, or at line 29. In both cases, the estimate sent is the estimate of some

process in phase 2 of round ρ . Thus some process must have set, in (a) phase 1 or (b) in a previous round, its estimate to a value that is not the proposal of any process. This is in contradiction with (a) the result established by case (1) or the assumption that ρ is the smallest round in which this happens.

Case (3). Assume that this happens at line 12 (either in phase 1 or phase 2): some process p_i sets for the first time est_i to a value that is not the proposal of some process. Process p_i must receive a $(\rho', ph_j, P_j, est_j)$ with $\rho' > \rho$ and such that est_j is not the proposal of some process. Since ρ is the smallest round in which some process adopts such an estimate, some process must have adopted est_j in round ρ , *i.e.*, either in case (1) or case (2) above, which have been proven to be impossible. Again we conclude by contradiction.