The Mutable Consensus Protocol*

J. Pereira
Universidade do Minho
jop@di.uminho.pt

R. Oliveira
Universidade do Minho
rco@di.uminho.pt

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 compromise 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. Moreover, their implementation can be highly simplified if based on a central consensus module [12], hence the relevance of the coverage and efficiency of consensus protocols. Nevertheless, a fundamental result states the impossibility of deterministic consensus in asynchronous distributed systems where at least one process may crash [8]. This impossibility can be circumvented by strengthening the asynchronous model with additional assumptions.

We focus on protocols based on unreliable failure detectors [7, 21] 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, 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 [21] 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 [7], 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 [2].

In this paper we address this issue: We present a consensus protocol allowing a 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 proposed algorithm results in good performance. By varying system parameters we ensure that performance gains can be obtained in a variety environments.

Our proposal is done in two steps. First, we present a new consensus protocol based on *stubborn channels* [10]. 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 imple-

* Supported by FCT, project STRONGREP (POSI/CHS/41285/2001).

mentation 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, number of messages and the number of bytes transmitted.

Such delays avoid the actual transmission of a message m 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 of m 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 [12]. 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 [7, 21]. 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.

In short, the paper makes two main contributions:

i. A mutable consensus protocol, that can be configured to a large variety of message exchange patterns with a simple correctness preserving technique that can be applied in run-time and without coordination.

ii. The permutation gossip mutation, which is a general purpose implementation of a stubborn channel that can be used to obtain an efficient and scalable consensus protocol.

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 $\{p_1, p_2, \ldots, p_n\}$. 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. Assuming an asynchronous system model means that the correctness of an algorithm holds despite the actual timing of the system. Using such model is justified to achieve robust protocols that are correct regardless of delays that cannot be (easily) controlled by the programmer, such as clock synchronization, message passing duration and scheduling of processes. In fact, in an asynchronous system model the resulting algorithms are correct also regardless of delays that can be controlled by the programmer. If desired, one can *insert finite delays* in the implementation of an algorithm and it would still be correct.

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 $\Diamond S$ [7] enabling us to circumvent the FLP impossibility result [8].

Processes are completely connected through a set of fair-lossy communication channels [3]. Each channel connecting process p_i to p_j is defined by a pair of primitives $\mathsf{Send}_{i,j}(m)$ and $\mathsf{Receive}_{j,i}(m)$. 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.

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 func-

tion Consensus and then decide on its return value such that the following properties hold [7]:

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 $\Diamond S$ [7, 21, 14]. 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 being able to broadcast an estimate, the new coordinator is required to also gather 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 [9]. Even if the failure detector misbehaves (*i.e.*, the assumption that the failure detector is of class $\Diamond S$ turns out to be wrong) the protocol ensures safety.

Protocols differ mostly in how majorities of votes are collected. In a centralized protocol [7], 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 [21] 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 [14]. 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.

Limitations on scalability can usually be mitigated by distributing the message load among 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 its 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 of stubborn communication channels [10].

3.1. Stubborn Channels

A stubborn channel [10] connecting two processes p_i and p_j is an unreliable communication channel defined by a pair of primitives $\mathsf{sSend}_{i,j}(m)$ and $\mathsf{sReceive}_{j,i}(m)$, 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. More precisely, we consider here a 1-stubborn channel. The general case, k-stubborn, requires the buffering and the retransmission of the k last messages.

3.2. Algorithm

The general approach to obtain a mutable protocol, which can be reconfigured to mimic protocols with different message exchange patterns, is as follows:

i. Obtain an algorithm that is correct in an asynchronous system model.

ii. Modify the algorithm to use *stubborn channels*, mostly by ensuring that it can skip messages that could have been discarded or duplicated.

iii. Ensure that the algorithm encompasses a large number of possible message exchange patterns by, as often as

Process p_i:

Function Consensus(v_i):

```
1  (est_i.val, est_i.proc) ← (v_i, i); r_i ← 1;
2  while true do
3     ph_i ← 1; P_i ← ∅; coord_i = (r_i mod n) + 1
4     if i = coord_i then
5        P_i ← {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 ≤ 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 ← est_j; r_i ← r_j;
13                ph_i ← ph_j; P_i ← ∅;
14             endif;
15             if r_i = r_j ∧ ph_i < ph_j then
16                ph_i ← ph_j; P_i ← ∅;
17             endif;
18             if (r_i = r_j ∧ P_j ⊈ P_i)∨
19                   (ph_j = 1 ∧ #P_j > n/2) then
20                P_i ← P_i ∪ P_j ∪ {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 ∧ ph_i = 1 then
28                ph_i ← 2; P_i ← {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 ← r_i + 1;
35 endwhile
```

Figure 1. Mutable consensus.

possible, broadcasting messages instead of directing them to a single destination and by relaying all information received, instead of relying on it being directly conveyed.

In Figure 1 we present a consensus algorithm obtained following this method. Apparently the result is not configurable and, even worse, has a message exchange pattern which is not attractive by any performance metric, especially when considering the number of messages exchanged. This is however a prejudgment based on intuition accustomed to reliable channels. In fact, sending messages through stubborn channels does not imply that those messages are actually transmitted through the underlying network, but only that eventually some message will be transmitted. It is thus possible that only a small fraction of the messages sent by the protocol actually reaches the network. In Section 4 we exploit precisely this fact.

The algorithm of Figure 1 proceeds in asynchronous rounds. Each round has a designated coordinator that tries to impose its proposal as the decision value. Each round has two phases. 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. If a process receives a message from a larger round it immediately jumps to that round.

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 the processes that p_i knows are endorsers of the current coordinator (phase 1) or the processes that are detractors of the coordinator (phase 2). In each round the coordinator sets itself as an endorser of the coordinator´s estimate and initiates the round broadcasting its estimate and its set of voters (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.

Lines 18 to 25 are actually the core of the algorithm. Before it, the code handles messages from larger rounds or phases, and lines 26 to 30 handle the suspicion of the current coordinator. A processes exits the main loop of each round (lines 8 to 32) whenever it collects a majority of endorsers or detractors.

In detail, 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 is 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). However, the two cases are not distinguished and are dealt with in the same way. When the received message is from the same round that p_i is in and brings new votes ($P_j \not\subseteq P_i$), 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

Process p_i:

State:
$b_{i,j}$ initially $\perp$
$t_{i,j}$ initially ∞

Function $\mathsf{sSend}_{i,j}(m)$:
1 $t_{i,j} \leftarrow \Delta_0(m)$;
2 $b_{i,j} \leftarrow m$;

Task $\mathsf{retransmit}_{i,j}$:
3 **while** $true$ **do**
4 **wait until** $t_{i,j} \leq \mathsf{Clock}_i$;
5 $\mathsf{Send}(b_{i,j})$;
6 $t_{i,j} \leftarrow \Delta$;
7 **endwhile**

Process p_j:

Function $\mathsf{sReceive}_{j,i}(m)$:
8 return $\mathsf{Receive}_{j,i}(m)$;

Figure 2. Stubborn channel from p_i to p_j.

a majority set of votes for phase 1 regardless of the round in which 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. Suspicions 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.

A more thorough description of the algorithm and a correctness proof can be found in the extended version of this paper [18].

4. Protocol Mutations

In the implementation of stubborn channels, an issue with crucial impact in practice, is the retransmission period. A very short delay will use excessive resources, while a long delay will introduce high latency when the network is lossy. Figure 2 presents a simple, slightly detailed, 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 finite, 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 reception primitive (line 8).

Process p_i:

Function $\Delta_0(m)$:
1 **if** $\mathrm{fresh(b_{i,j}, m)} \vee \mathrm{maj}(m)$ **then**
2 return Clock_i;
3 **else**
4 return $\mathsf{Clock}_i + e$;
5 **endif**;

Function Δ:
6 return $\mathsf{Clock}_i + e$;

Figure 3. "Early" mutation.

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 2 just by instantiating functions $\Delta_0(m)$ and Δ. This trivially ensures their correctness. We admit that these functions can read the local state associated with the channel, *i.e.*, $t_{i,j}$ (the timeout from i to j) and $b_{i,j}$ (the buffer from i to j).

Some of these mutations use the semantics of messages exchanged by the consensus algorithm. The computation of delays involves evaluating the following conditions of buffered messages:

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

For any consensus message $m = (r, ph, P, est)$ we have $\mathrm{round}(m) = r$, $\mathrm{phase}(m) = ph$ and $\mathrm{voters}(m) = P$.

Early The *early* mutation implementation of Figure 3 is the simplest and aims at a message exchange pattern similar to that of early consensus [21], 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 must 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 the current value of 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 to avoid the actual transmission of the message. This makes it unlikely that all messages but the first from the coordinator and the last (with a majority of votes) are actually sent obtaining the desired result.

Centralized The *centralized* mutation implementation of Figure 4 aims at producing a message exchange pattern which resembles that of the Chandra-Toueg centralized algorithm [7]. In contrast to the *early* mutation, this one does

Process p_i:

Function $\Delta_0(m)$:
1 $c \leftarrow (round(m) \mod n) + 1$
2 **if** $((i = c \vee j = c) \wedge \text{fresh}(b_{i,j}, m)) \vee \text{maj}(m)$ **then**
3 return Clock_i;
4 **else**
5 return $\mathsf{Clock}_i + e$;
6 **endif**;

Function Δ:
7 return $\mathsf{Clock}_i + e$;

Figure 4. Centralized mutation.

Process p_i:

Function $\Delta_0(m)$:
1 **if** $j = ((i+1) \mod n) + 1 \wedge (\text{fresh}(b_{i,j}, m) \vee \text{maj}(m))$ **then**
2 return Clock_i;
3 **else**
4 return $\mathsf{Clock}_i + e$;
5 **endif**;

Function Δ:
6 return $\mathsf{Clock}_i + e$;

Figure 5. Ring mutation.

not reproduce exactly the original protocol, as the coordinator does not gather estimates when it enters a round. In fact, it differs from the *early* mutation only by avoiding the direct transmission of votes among participants. Instead, these are relayed by the coordinator.

Therefore, if the sender or the receiver process is the current coordinator, the same delays of the *early* implementation are used. Otherwise, messages are delayed (line 5) by the estimated time required to end consensus, thus avoiding their transmission. Decision messages 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 $j = ((i+1) \mod n) + 1$. 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. As soon as a message is sent, 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. It is worth noting that, in contrast with

Process p_i:

State:
c initially 0
u initially a permutation of $1 \ldots n$

Function $\Delta_0(m)$:
1 $v \leftarrow \mathsf{turns}_F(u, c, j)$;
2 $c \leftarrow c + v + 1$;
3 return $\mathsf{Clock}_i + ve$;

Function Δ:
4 $v \leftarrow \mathsf{turns}_F(u, c, j)$;
5 $c \leftarrow c + v + 1$;
6 return $\mathsf{Clock}_i + (v+1)e$;

Function $\mathsf{turns}_F(u, c, j)$:
7 $x \leftarrow 0$;
8 **while** $\nexists l : 0 \leq l < F \wedge$
9 $u[((l + F(c + x)) \mod n) + 1] = j$ **do**
10 $x \leftarrow x + 1$;
11 **endwhile**;
12 return x;

Figure 6. Permutation gossip mutation.

other mutations, 1) all message delays are computed using the same rule and that this rule does not depend on the particular message contents, and 2) messages are not equally delayed to all destinations. In fact, with respect to 2), 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 $\mathsf{turns}_F(u, c, j)$ 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 identically (lines 4 to 6).

5. Evaluation

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

5.1. Implementation

The implementation of the mutable consensus protocol and stubborn channels used for evaluation was done in Java. A single thread is used for each process and 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 stored 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 [15]. Therefore we use a centralized simulation model [1] 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 timelines 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 [1] 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.

The centralized simulation runtime used is also implemented in Java and uses a virtual per-process CPU cycle counter to measure time [20]. 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 [1].

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 higher priority is assigned to the protocol task.

The simulated network mimics a switched 10Mbit/s 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 the 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.

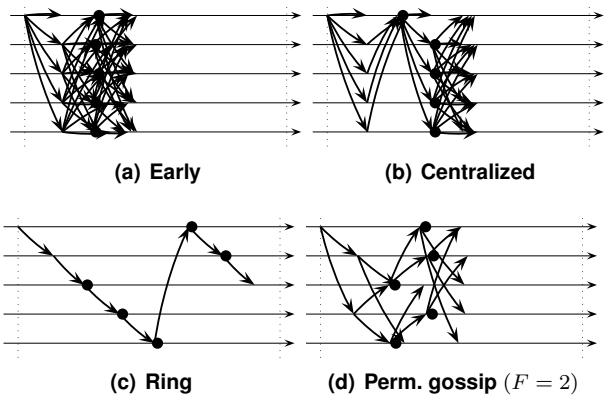


Figure 7. Prefixes of typical executions (1ms).

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.

5.4. Results

A first intuition on the impact of the delays introduced in each mutation 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 the stubborn 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 pro-

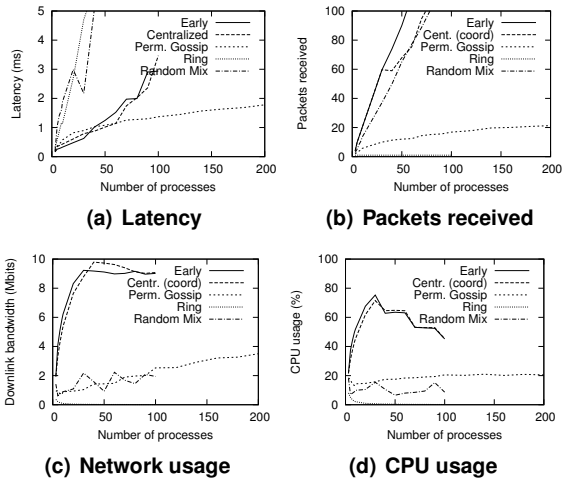


Figure 8. Performance of protocol mutations.

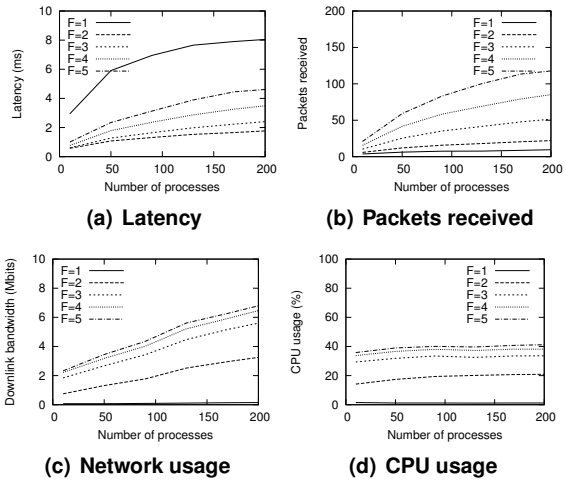


Figure 9. Protocol parameters.

posal 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 8(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 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 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. 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 and thus its good 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 [7, 21, 17, 14, 11] rely on a similar message exchange pattern in which at least one process receives messages from all others.

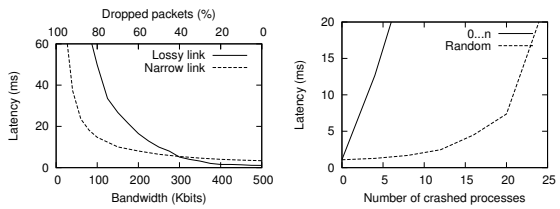
The *ring* mutation is extremely frugal in terms of resource consumption, although resulting in high latency. It would however be desirable if a large number of consensus instances would be running simultaneously and latency is not a primary concern. Finally, the *permutation gossip* is scalable to a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

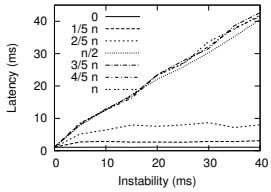
6. Configuration of Permutation Gossip

6.1. Protocol Parameters

It is clear from the previous section that the *permutation gossip* mutation is the most interesting, in particular, with a large number of participants. The performance of the *permutation gossip* mutation is intimately related to 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 ob-



(a) Variable bandwidth (b) Crashed processes



(c) False suspicions

Figure 10. Environment conditions.

tained 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 confirmed that, with $F = 2$ and e larger than the expected duration of a full run, the performance of the protocol in normal conditions is independent of e, as no retransmission needs to be performed to terminate.

6.2. Environment Conditions

Gossip-based protocols are known for their resilience to network omissions and process crashes. By using the centralized simulation facility we can easily crash processes, inject false suspicions, random message 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). 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 1Mbit/s (see Figure 9(c)), the protocol is still functioning reasonably well with as little as 250Kbit/s and collapses only with less than 50Kbit/s.

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. However, it is not as serious as process crashes since 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 wrongly suspecting 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.* [4, 6]). 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 [14] and the oracle used [11]. The first approach [14] 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 [11] 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 [5] and stability detection [13]. 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 [16], 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 simple to 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, compromise the correctness of the 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 consider using information from the environment (*e.g.* network conditions). Finally, no coordination at all is required when selecting the strategy used to compute delays.

Albeit simple, the *permutation gossip* introduced in this paper performs very well. 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 [19]. 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] G. Alvarez and F. Cristian. Simulation-based testing of communication protocols for dependable embedded systems. *The Journal of Supercomputing*, 16(1), May 2000.

[2] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *ACM Symp. Principles of Distributed Computing*.

[3] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Intl. Ws. Distributed Computing (WDAG)*, 1996.

[4] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *ACM Symp. Principles of Distributed Computing*, 1983.

[5] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Computer Systems*, 17(2), 1999.

[6] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4), July 1996.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), March 1996.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, April 1985.

[9] R. Guerraoui. Indulgent algorithms. In *ACM Symp. Principles of Distributed Computing*, 2000.

[10] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98-278, EPFLausanne, June 1998.

[11] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. Technical Report PI-1531, IRISA, April 2003.

[12] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. Software Engineering*, 27(1), January 2001.

[13] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K. Birman. GSGC: An efficient gossip-style garbage collection scheme for scalable reliable multicast. Technical Report TR97-1656, Cornell University, CS Department, December 1997.

[14] M. Hurfin, A. Mostefaoui, and M. Raynal. A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Trans. Computers*, 51(4), April 2002.

[15] I. Keidar. Challenges in evaluating distributed algorithms. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*. Springer, 2003.

[16] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In *IEEE Intl. Ws. Applied Reliable Group Communication*, 2001.

[17] R. Oliveira. *Solving Consensus: From Fair-Lossy Channels to Crash-Recovery of Processes*. PhD thesis, EPFLausanne, February 2000.

[18] J. Pereira and R. Oliveira. The mutable consensus protocol (extended version). Technical report, University of Minho, August 2004.

[19] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast: Definition, implementation and performance evaluation. *IEEE Trans. Computers*, 52(2), February 2003.

[20] M. Pettersson. Linux performace monitoring counters. http://www.csd.uu.se/ mikpe/linux/perfctr/.

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