Worldwide Consensus

Francisco Maia, Miguel Matos, José Pereira, Rui Oliveira

▶ To cite this version:

HAL Id: hal-01583578

https://inria.hal.science/hal-01583578

Submitted on 7 Sep 2017

Worldwide Consensus

Francisco Maia, Miguel Matos, José Pereira, and Rui Oliveira [*]

High-Assurance Software Laboratory
University of Minho
Braga, Portugal
{fmaia,miguelmatos,jop,rco}@di.uminho.pt

Abstract. Consensus is an abstraction of a variety of important challenges in dependable distributed systems. Thus a large body of theoretical knowledge is focused on modeling and solving consensus within different system assumptions. However, moving from theory to practice imposes compromises and design decisions that may impact the elegance, trade-offs and correctness of theoretical appealing consensus protocols.
In this paper we present the implementation and detailed analysis, in a real environment with a large number of nodes, of mutable consensus, a theoretical appealing protocol able to offer a wide range of trade-offs (called mutations) between decision latency and message complexity. The analysis sheds light on the fundamental behavior of the mutations, and leads to the identification of problems related to the real environment. Such problems are addressed without ever affecting the correctness of the theoretical proposal.

1 Introduction

The problem of fault-tolerant consensus in distributed systems has received much attention throughout the years as a powerful abstraction at the core of several practical problems, namely atomic commitment, atomic broadcast and view synchrony. Furthermore, the variety of models in which consensus can be solved led to the appearance of several protocols targeted at system models with different assumptions on the synchrony of processes and communications channels, on the admissible failure patterns, and on failure detection.

For the numerous consensus protocols present in the literature, a generic differentiator, with major relevance in practical terms, is the network-level communication pattern that emerges from each particular design. For instance, in Chandra and Toueg's centralized protocol [?], a rotating coordinator process is in the center of all communication: the coordinator sends its proposal to all other participants, then collects votes from everyone and finally broadcasts the decision. A different approach is taken in Schiper's Early Consensus protocol [?] where all participants always broadcast their messages. Different protocols thus

---

present different communications patterns with distinct message complexity and communication steps that establish several trade-offs on the decision latency, network usage, resilience to message loss and processor load.

From a practical point of view, this implies that the system architect needs to carefully choose a protocol suited to her specific requirements. In dynamic environments where requirements change, this most probably requires the selection of several protocols and to change implementations as required.

An attractive alternative is the Mutable Consensus protocol [**?**]. In short, mutable consensus can seamlessly adjust the trade-offs on network usage, processing load and fault tolerance through a range of mutations encapsulated in the protocol itself. More strikingly, these mutations lie outside the system model used to prove the algorithm's correctness. That is, the whole protocol has been proved correct regardless of the mutations being adopted by any of the participant processes. This makes Mutable Consensus specially appealing to dynamic environments where application requirements may change and also useful in practice as one needs only to implement and test a single solution that is equivalent to a range of typical consensus protocols.

However, making the algorithm presented in [**?**] an executable implementation has been a challenging task. More specifically, making it capable of being used as a generic consensus module and evaluating it in a real world, heterogeneous and fairly large distributed system uncovered several non-trivial issues. These stem mainly from simplifications and omissions common at the modeling abstraction level but that have a profound impact in a real implementation. This divergence between reality and model perfection seems to pervade implementation efforts as also noted in similar endeavors [**?**].

This paper presents the implementation and analysis of the Mutable Consensus protocol in the PlanetLab [**?**] environment. Our contributions are: i) the implementation itself, ii) the identification of several problems raised when going from a high-level specification to executable code, iii) the solutions and workarounds for these problems, and finally iv) an insight on the trade-offs offered by each of the protocols mutation in a large distributed system.

The paper is organized as follows: Section 2 presents a succinct description of the Mutable Consensus protocol and the Stubborn Communication Channels [**?**] upon which it depends. Section 3 describes the implementation process and the concrete resulting code. Section 4 details the protocol evaluation in PlaneLab and interprets the results based on the characteristics of the different mutations. Section 5 concludes the paper.

2   Mutable Consensus

The Mutable Consensus protocol [**?**] solves the consensus problem [**?**] tolerating the crash of a majority of processes. It assumes an asynchronous distributed system model augmented with an eventual strong failure detector, $\Diamond\mathcal{S}$ [**?**]. Processes are considered to be fully connected through fair-lossy communication channels. A fair-lossy channel closely models existing network links requiring the weakest

reliability properties to still be useful: any message that is sent has a non-zero probability to be delivered. Over these channels, the mutable protocol leverages a simple yet powerful abstraction given by Stubborn communication [?].

In the following we recall the definitions of consensus and Stubborn Channels and provide an overview of the Mutable Consensus protocol.

2.1   The Consensus Problem

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. All processes are expected to start the protocol with an undecided value for the decision and proposing some value through function Consensus. Each *correct* process, that is a process that does not crash, is expected to finish the protocol as soon as it decides on a value such that the following properties hold [?]: **Validity** If a process decides v, then v was proposed by some process; **Agreement** No two processes decide differently and **Termination** Every correct process eventually decides some value.

2.2   Stubborn Communication Channels

A Stubborn Channel [?] 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.

Intuitively, a stubborn channel adds to the reliability of a fair-lossy channel by strengthening the delivery guarantees of the *last* message that is sent. As soon as a message is sent, it makes the previous one obsolete. Stubborn channels were initially proposed as a way to reduce the buffer footprint required by reliable communication [?] but in the Mutable Consensus protocol they are the key to the algorithm mutations regarding the network-level usage patterns. The stubborn property allows messages to be lost and the Mutable Consensus takes advantage of it to have only a subset of the messages effectively sent over the network.

To implement a stubborn channel over a fair-lossy channel it suffices to buffer the last message sent and retransmit it periodically.

2.3   Protocol Description

Like most agreement protocols based on the asynchronous distributed system model, the Mutable Consensus protocol uses the rotating coordinator paradigm and proceeds in asynchronous rounds. Each round has two phases. In the first phase of some round, processes try to agree on the coordinator's proposal for the decision. If the coordinator is suspected to have failed, then the second phase starts and processes try to agree to detract the current coordinator and proceed to the next round.

In both phases agreement is reached as soon as a majority of processes share the same opinion. For each round, each process keeps a list of processes that currently have a similar opinion, which can be either supporting the coordinator on the current value proposed (phase 1), or retracting the coordinator when suspecting it failed (phase 2). In the protocol, communication is used to *broadcast* these lists among the participating processes. Stubborn communication channels handle the communication in ways such that, at the network-level, distinct message patterns emerge, as if the consensus protocol itself actually mutates.

A first look at the Mutable Consensus protocol 1 hints at a similar message exchange pattern to that of the *Early Consensus* protocol [?] which is not attractive due to the probable redundancy of the messages' contents and the quadratic complexity of the exchange pattern (all processes send their lists to all). However, from the protocol specification and the stubborn property of the communication channels, it is possible that only a subset of the messages are actually transmitted over the network. Firstly, of the messages sSent by the protocol only messages with *new* information are actually broadcast. Then, at the stubborn channel level, not all of those messages are readily sent, they are judiciously delayed in such a way that, in *good runs* they become obsolete and end up not being transmitted at all. As introduced in [?] and detailed in the next Section, a sensible implementation of the Stubborn Channels can match the subset of transmitted messages with the minimum set of messages needed to reach consensus. This is achieved by configuring different send delays, which allow to radically alter the message exchange pattern without ever impacting the protocol's correctness. These configurations are called protocol mutations.

Four mutations have been proposed: early, centralized, ring and gossip. The early mutation assigns to each message zero delay. This enforces an actual broadcast of each message and the protocol behaves as expected at higher level. For the other mutations, some messages will be sent immediately while others only after a period of time e, which is an estimate on the time consensus will take and therefore sending those messages is expected to be avoided. Following this idea, in the centralized mutation, only messages to and from the coordinator process are immediately sent while the others are delayed. In the ring mutation only messages addressed to the next process (in a logical ring) are immediately sent. Finally, for the gossip mutation, each process has a permutation of the list of all processes and sends the message immediately to f processes (gossip fanout) and delays it to the others. Parameter f is configurable and this set of processes changes for each broadcast.

It is important to notice that delayed messages are never discarded. In fact, if, after e elapses, the subset of messages transmitted was not enough to reach consensus the Stubborn Channel will send those messages allowing the protocol to make progress. Moreover, in the original proposal this actually means that all the mutations may degenerate into the early mutation. This observation is discussed in Section 3.2.

3 Mutable Consensus Made Live

A complete implementation of the Mutable Consensus protocol, capable of running in a real large scale environment, uncovered some challenges previously not considered at the theoretical level. These challenges not only raised practical issues but also led us to propose some changes in the algorithm itself, namely in the various mutations definition.

The implementation was done using the Splay [?] platform and the Lua programming language. Splay enables the specification of distributed algorithms in a very concise way using Lua, and enables the deployment in a number of different testbeds including PlanetLab [?]. The ability to deploy in PlanetLab allows the use of a number of nodes not available to us at the laboratory. Moreover, the real environment helps to test the application against different unpredictable network and node failures.

After a careful study of the original algorithm three main challenges arose, namely in the implementation of the core of protocol, in the implementation of the stubborn channels, and in the achievement of quiescence.

3.1 Mutable core

Splay is an event-driven framework where processes communicate through remote procedure calls (RPC). To avoid blocking RPC calls and Mutable Consensus is message based, threads are used to parallelize such invocations. This improves the performance of the algorithm and matches the original definition of the protocol. The event loop is started by events.run(f) which invokes function f and waits from incoming events received by means of RPCs. Processes terminate by calling events.exit(). Each process has a list $plist$, containing the identifier of all n participants and is identified by its position on that list, given by pId.

The implementation of the Mutable Consensus is presented in Listing 1. It closely resembles that of [?]. Initially, the consensus() function is called, which begins the event loop (lines 3∼11) and calls start. In function start (lines 13∼23), the coordinator, given by $((r_i \bmod n) + 1)$, initiates the protocol by calling sSend with the following parameters: its identifier pId, the round r_i and phase 1, the list of supporters $voters$ (currently itself) and the estimate est for all nodes (line 20). sSend is presented later in Listing 2.

The protocol then proceeds by exchanging messages, which correspond to sReceive calls. Upon the reception of a message a process proceeds as follows:

– If its list of supporters, $voters$, does not contain a majority of votes yet then it evaluates the received message as follows: if the message comes from a larger round r_j then the process jumps to that round and resets the list of supporters (lines 27∼32); if the message belongs to the current round but to a larger phase, then the coordinator has been suspected and thus the process changes phase and starts collecting a majority of detractors (lines 33∼36); otherwise if the message belongs to the current round and contains new votes, or it is from phase 1 and already contains a majority of votes

```lua
1    consensus_decision = nil
2
3    function consensus(value)
4      --Global State
5      voters = Set.new{}
6      est = {val=value,proc=pId}
7      ri = 1
8      phi = 1
9      --starts event loop
10     events.run(function () start() end)
11   end
12
13   function start()
14     if pId == ((ri mod n) + 1) then
15     -- the coordinator initiates the protocol by
16     -- broadcasting its estimate
17       voters = Set.new{pId}
18       est.proc = pId
19       for k in plist do
20         sSend(k,{pId,ri,phi,voters,est})
21       end
22     end
23   end
24
25   function sReceive(s,{rj,phj,votersj,estj})
26       if Set.len(p) <= n/2 then
27         if ri<rj then
28             est = estj
29             ri = rj
30             phi = phj
31             voters = Set.new{}
32         end
33         if (ri==rj and phi<phj) then
34             phi = phj
35             voters = Set.new{}
36         end
37         local newVotes = (ri==rj) and ( not Set.isContained(votersj,voters) )
38         local majority= (phj==1) and (Set.len(votersj)>n/2)
39         if (newVotes or majority) then
40             voters = Set.union(voters,Set.union(votersj,Set.new{pId}))
41             if estj.proc == ((ri mod n) + 1) then est = estj end
42             for k in plist do
43                 sSend(k,{pId,ri,phi,voters,est})
44             end
45         end
46       end
47       if Set.len(voters) > n/2 then
48           if phi == 1 then
49             consensus_decision = est.val
50             events.exit()
51           else
52             ri = ri + 1
53             phi = 1
54             voters = Set.new{}
55           end
56       end
57   end
58
59   function suspected(j)
60     if j == ((ri mod n) + 1) and phi == 1 then
61       phi = 2
62       voters = Set.new{pId}
63       for k in plist do
64         sSend(k,{pId,ri,phi,voters,est})
65       end
66     end
67   end
```

Listing 1. Mutable consensus implementation in Lua

endorsing the coordinator's estimate, then the process updates its set of supporters *voters* and broadcasts it (lines 37∼45).
- If its list of supporters, *voters*, already contains a majority of votes (lines 47∼56) then, if phase is 1 (endorsing the coordinator) the process decides and exits, otherwise (phase 2, detracting the coordinator) the process moves to the next round.

Finally, should the coordinator become suspected (lines 59∼67), the process immediately changes to phase 2 and broadcasts its suspicion to force a change of round. Function suspected is invoked by the failure detector module not detailed in the paper.

3.2   Stubborn channels

From the definition, a Stubborn Channel implementation should be fairly straightforward but, nonetheless, some subtleties arose. This section identifies those issues and describes the proposed solutions.

Implementation. A Stubborn Channel requires primitives sSend(k, m) and sReceive(m) where k is the destination process and m is the message. sReceive (Listing 1) has no special semantics and is given by Splay's RPC mechanism.

sSend is presented in Listing 2 (lines 1∼6). The actual *send* of the message is done in line 6 by remotely invoking, through Splay, the destination's sReceive function. The sSend requires two auxiliary functions: delta0/delta which are responsible for the protocol's mutations, and retransmission which handles the periodic retransmission of messages to overcome message loss. When a message is sSent, it is buffered in bstate (line 3) and, if delta0 determines so, it is sent immediately to the network (lines 4∼6). Otherwise, it will wait to be handled by the retransmission function.

Unlike the original algorithm that handles the retransmission of messages per channel separately, our implementation, for the sake of scalability, deals with all open channels in batch through a single thread that animates the function retransmission. Periodically, function retransmission() (lines 10∼24) determines for each destination k the need to send bstate[k] by means of delta(k). Variables *tries* and *maxtries* will be discussed in Section 3.2.

The delta0(k, m) and delta(k) functions determine if a message is to be sent immediately or delayed. delta0(k, m) is used for the first time a message is sent and delta(k) in message retransmission. It is important to notice that delaying a message does not compromise in any way the guarantees given by the Stubborn Channel. By delaying certain messages and immediately sending others the message exchange pattern is altered. Different implementations of delta0(k, m) and delta(k) yield different mutations. The implementation of the four original mutations are presented in Listing 3.

In the early mutation, Listing 3(a), delta0(k, m) always return *true* for *useful* messages. Useful messages either contain a majority of votes (maj(m)) or are new

```
1   function sSend(k,m)
2     t = delta0(k,m)
3     bstate[k] = m -- holds the messages to all destinations
4     if t then
5       -- invoke sReceive in a separate thread
6       events.thread(function() rpc.acall(destination[k],{"sReceive",m}) end)
7     end
8   end
9
10  function retransmission()
11    return events.thread(
12      function()
13        tries = 0
14        while true do
15        events.sleep(defaultDelta)
16        for k in p do
17          if delta(k) or tries > maxtries then
18            rpc.acall(destination[k],{"sReceive",bstate[k]}
19          end
20          tries = tries + 1
21        end
22      end
23    end)
24  end
```

Listing 2. Stubborn Channels implementation in Lua

($\mathsf{fresh}(m1, m2)$). In this mutation, $\mathsf{delta}(k)$ always returns true. This will enforce an actual broadcast by having all messages transmitted over the network.

The centralized mutation, Listing 3(b), requires a slight change. Only messages to and from the coordinator are immediately sent. This will enforce a centralized message exchange pattern where processes send all the votes to the coordinator. When the coordinator gathers a majority it broadcasts the decision.

The ring mutation, Listing 3(c), is similar to the previous one except that only messages to the next process, in a logical ring, are immediately sent. The protocol will act as if nodes were physically connected in a ring topology.

The gossip mutation presented in Listing 3(d) intends to offer the high scalability properties of gossip-based protocols, which should allow Mutable Consensus to scale to a large number of nodes. Each process keeps a permutation u of the list of processes. Each time a message is sent it is immediately transmitted to the next f (fanout) processes in the list. Variable c (Listing 2) is used to vary the list of f destination processes. This variable is incremented each time a broadcast is invoked.

Real runs of the algorithm yielded by configuring the mutable consensus protocol with the various mutations are depicted in Figure 1. Each horizontal line represents a process, arrows the messages, and black dots the decision.

Mutation Degeneration. After running the protocol several times we observed that the ring and centralized mutations may degenerate into the early mutation. Degeneration means that retransmissions end up being made immediately without regard to the mutation.

```
1  function delta0 (k ,m)
2    return ( fresh ( bstate [ k ] ,m) or maj (m))
3  end
4
5  function delta (k ,m)
6    return true
7  end
```

(a) Early

```
1  function delta0 (k ,m)
2    coord = ( ri mod n) + 1
3    return (k==coord or pId==coord )
4      and ( fresh ( bstate [ k ] ,m) or maj (m))
5  end
6
7  function delta (k ,m)
8    return true
9  end
```

(b) Centralized

```
1  function delta0 (k ,m)
2    return (k == (( pId % n) + 1))
3      and ( fresh ( bstate [ k ] ,m) or maj (m))
4  end
5
6  function delta (k ,m)
7    return true
8  end
```

(c) Ring

```
1   u = perm(n); c = 1; turn = 0; f = 4
2   function delta0 (k ,m)
3     return delta (k)
4   end
5
6   function delta (k)
7     turn = turn + 1
8     if turn == n then
9       c = c+f; turn = 0; end
10    local l = 0
11    while ( l<f ) do
12      if u [( l+c) % n] == k then
13        return true end
14      l = l + 1
15    end
16    return false
17  end
```

(d) Gossip

Listing 3. Mutations implementation in Lua

(a) Early

(b) Centralized

(c) Ring

(d) Gossip (F=5)
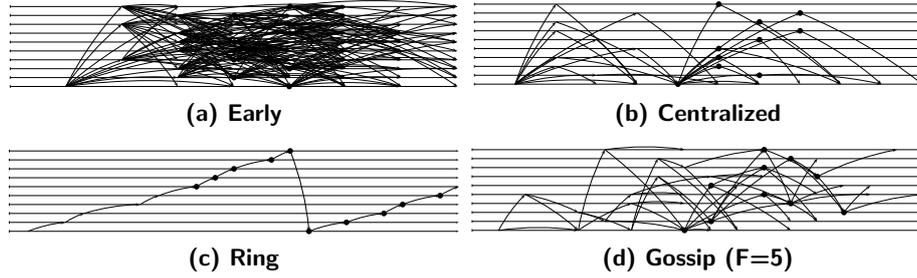
Fig. 1. Prefixes of typical executions.

In each sSend messages are divided into two groups: those immediately sent and those that will be delayed. The first ones will follow the pattern defined by the mutation while the others are expected never to be sent as consensus is reached before $defaultDelta$ (the estimate on the time consensus will take to finish, e in Section 2.3) expires.

However, if something goes wrong in the first round, such as node failures or message loss, all delayed messages will be sent at once as a result of the delta(k) implementation that always returns *true* for these mutations. This implies degenerating into the early mutation as depicted in Figure 2(a) for the ring mutation. Similar behavior is observed for the centralized mutation.

(a) Degenerating mutation.   (b) Mutation preserved
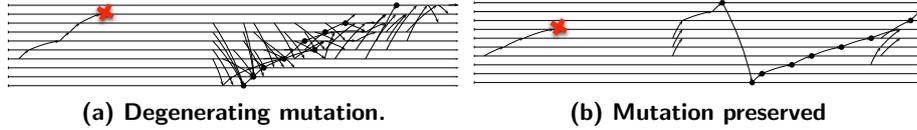
Fig. 2. Mutation degeneration correction.

```
1    function delta(k)
2        return (k == ((myposition % n) + 1))
3    end
```

Listing 4. Improved function delta for the ring mutation

Essentially, the degeneration happens because after the $defaultDelta$ period all messages are treated equally. Therefore, we modified the delta function to take this issue into account, and selectively send some messages and further delay the others. The delta function now becomes similar do $delta0$ without the $fresh$ check as messages are being retransmitted and thus are not new, and the $majority$ check as the message needs to be retransmitted even if it does not hold a majority. The new delta function for the ring mutation is presented in Listing 4. Changes for the centralized mutation are similar and thus omitted.

The process is repeated a finite number of times after which the mutation *must* degenerate into the early mutation. In fact, if that was not the case, correctness could be compromised as some messages would never be sent. To overcome this, we additionally check if the number of allowed retransmissions (stored in variable $maxtries$) has been reached (Listing 2, line 17).

On top of these observations, the ring mutation revealed another interesting problem when deployed in real settings. In fact, maintaining the same order of the ring across rounds is not resilient to message loss. For instance, if the link between two nodes is prone to large message loss, consensus would only be reached when the ring mutation degenerates into the early mutation. This can be overcome by changing the ring on every round by simply computing the next process in the ring as follows: $((myposition + r_i)\%n) + 1$ where n is the number of processes and r_i the round number.

Quiescence When a process decides and terminates (Listing 1, lines 48 and 49) the last message it has broadcast corresponds to phase 1, contains a majority of votes in p and the decision value in est. To execute line 48, ph_i needs to be 1 and $|p| > n/2$. This can only happen if the process executed line 39 and got a majority of votes in p, since any other previous conditions in lines 26 or 32 result in $p = \{\}$. By the stubborn property of the communication channels all processes that do not crash will eventually receive this message and will, in turn, decide and terminate if they have not done so yet. Therefore the termination property of Consensus is satisfied.

However, any process that decides needs to keep the retransmission of its last message to ensure it is delivered. This means that, in this case, albeit the consensus instance terminates the process does not become quiescent. Given the recurrence of the algorithm this can become a problem as buffers from stubborn channels cannot be discarded and retransmission would go endlessly.

Achieving quiescent reliable communication with common failure detectors would require us to assume an eventual perfect failure detector ($\Diamond\mathcal{P}$)[**?**] which is stronger than needed to solve consensus and whose properties are much more difficult to attain in practice. To work around this problem, Aguilera et al. [**?**] have proposed the *heartbeat* failure detector $\mathcal{HB}$. Roughly, a $\mathcal{HB}$ failure detector provides each process with non-decreasing heartbeats of all the other processes and ensures that the heartbeat of a correct process is unbounded while that of a crashed process is bounded. Its implementation, in our model, is pretty simple: Each process periodically sends a heartbeat message to all its neighbors; upon the receipt of such a message from process q, p increases the heartbeat value of q. By combining the output of the $\mathcal{HB}$ failure detector with a simple positive acknowledgement protocol between the sSend and sReceive primitives we made the stubborn communication quiescent.

4   Evaluation

We evaluate the implementation of Mutable Consensus in a PlanetLab [**?**] environment by means of the Splay platform [**?**]. Splay was chosen because it allows the specification of algorithms in a very concise way using Lua, and enables the deployment in several testbeds including PlanetLab. The user simply specifies the number of nodes and Splay deploys the protocol in those nodes. A deployment for a run with 300 nodes is presented in Figure 5. The geographically dispersion and heterogenous nature of PlanetLab helps to test the application against different unpredictable network and node failures. Presented results are the average of 5 runs where each run represents a new Splay deployment.

A centralized logger gathers information about events. Due to asymmetries in nodes and links, events reaching the logger may deviate from the actual run. However, as results focus on comparison among runs, the conclusions stay valid.

Evaluation focus on two perspectives: consensus latency and message complexity. **Consensus Latency** is the time taken for processes to decide. We define two different metrics: *Coordinator Latency*, which is the time it takes for the coordinator to decide; and *Majority Latency*, which is the time it takes for a majority of nodes to decide.

With respect to latency the results obtained are depicted in Figure 3. The first remark has to do with the fact that the ring mutation did not produce results on runs with more that 50 nodes. In fact, in such runs the probability of message loss is very high and the ring easily breaks. To have a fair comparison, results are obtained from good rounds (before retransmissions) where the difference between mutations is clear. From the perspective of the coordinator, the centralized mutation exhibits higher latency than the earlier one. At first

(a) Coordinator Latency.   **(b) Majority Latency .**

Fig. 3. Consensus Latency.

(a) Number of messages coordinator.   **(b) Total number of messages.**
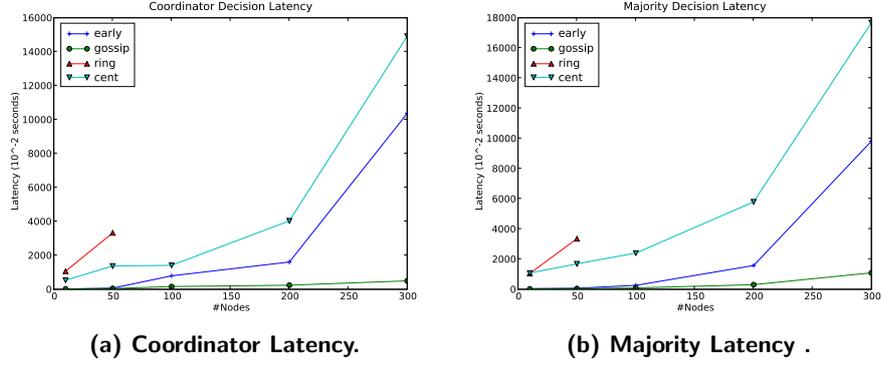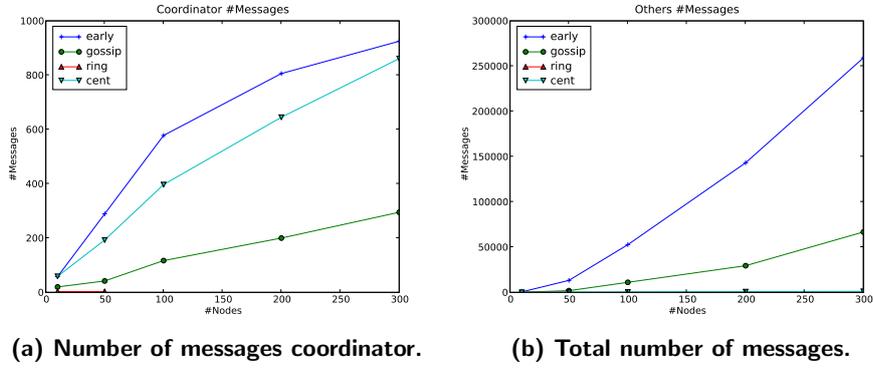
Fig. 4. Number of messages exchanged.

this seems unsettling. In fact, for the coordinator, in both mutations there are only two communication steps before decision: the coordinator's broadcast and then the collection of processes' votes. In spite of this similar behavior, in the centralized mutation the coordinator has to sequentially handle $n/2$ messages in all situations while in the early mutation each node can aggregate votes and propagate them to the coordinator. Considering PlanetLab's link asymmetry, it is likely that faster intermediate nodes propagate messages with a group of votes already gathered, which lowers coordinator's decision latency.

From the perspective of a majority of nodes, the most significant change is the gap between the centralized and early mutations. This is expected as the centralized mutation needs an extra communication step for the majority of nodes to decide.

It is important to notice that, both from the perspective of the coordinator and a majority of nodes, the centralized and early mutation have an abrupt increase in latency after runs with 200 nodes. This indicates a system saturation

Fig. 5. Node geographic distribution.

and an impediment to scalability of the protocol when configured with such mutations. On the other hand, the gossip mutation is virtually unaffected by the increase in the number of nodes. More interestingly, the gossip mutation is able to offer small latencies when compared to the other mutations. This stems from the inherently small number of hops each messages need to take to reach all nodes in a epidemic setting [**?**], and due to message frugality analyzed next.

Message complexity measures the network load each mutation implies. We defined two metrics: *Coordinator Messages*, which is the number of messages sent and received by the coordinator and *Others Messages* which is the total number of messages sent and received by other nodes, on average.

The results are depicted in Figure 4. The ring mutation clearly exchanges fewer messages in both metrics at the cost of higher latency. From the perspective of the coordinator, the early mutation and centralized mutations have a similar behavior. This is expected has the coordinator has to receive and send messages from and to all the participants. However, globally, the centralized mutation exchanges a considerably smaller number of messages at the cost of the extra communication step needed for the coordinator to broadcast its decision.

The interesting result is the gossip mutation. This mutation exchanges a more stable number of messages independently of the number of nodes and the metric. This is actually the key characteristic that enables this mutation to scale to a large number of nodes. As message exchanged is balanced across all the nodes the overall load is smaller. These results support the claim that the Mutable Consensus protocol is able to adapt to different environments and able to scale to a large number of nodes when configured with the gossip mutation.

5   Discussion

This paper described the implementation of the Mutable Consensus protocol in a real environment. The gap between theory and practice became evident as several non-trivial problems emerged. As in [**?**], those stem mainly from several simplifications that remained hidden both in the theoretical models and in the simulation tools used. Those issues have been addressed from a practical point

of view but without ever compromising correctness. As relevant additions to the algorithm we point out the avoidance of mutation degenerations and the quiescence of the stubborn communication channels.

To the best of our knowledge, this is the first work to analyze the behavior of a consensus protocol in an large scale hostile environment such as PlanetLab. With the gossip mutation, we have shown that the Mutable Consensus protocol can scale up to 300 nodes without compromising decision latency. This contrasts with the common belief that uniform consensus does not scale.

Mutable Consensus is adaptable by design simply by using different protocol mutations which makes it an attractive tool to solve consensus in a wide range of environments. Despite being adaptable, Mutable is not adaptive. However as it possesses the required properties to build a self-tuning system [**?**], offering Mutable Consensus as a generic self-contained software package is an important pursuit as future work. This could allow developers to use a modular and generic consensus service, averting the mix of different code components [**?**].

References

1. M. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29:2000, 2000.
2. A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Symposium on Networked Systems Design and Implementation*, pages 19–19, 2004.
3. T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Symposium on Principles of distributed computing*, pages 398–407, 2007.
4. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.
5. P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. From Epidemics to Distributed Computing. *IEEE Computer*, 37(5):60–67, May 2004.
6. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
7. R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn Communication Channels. Technical report, EPFL, 1998.
8. L. Leonini, E. Riviere, and P. Felber. SPLAY: Distributed Systems Evaluation Made Simple. *Symposium on Networked systems design and implementation*, pages 185–198, 2009.
9. M. Matos, J. Pereira, and R. Oliveira. Self Tuning with Self Confidence. In *"Fast Abstract", International Conference on Dependable Systems and Networks*, 2008.
10. J. Pereira and R. Oliveira. The mutable consensus protocol. In *Symposium on Reliable Distributed Systems*, pages 218–227, 2004.
11. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10:149–157, April 1997.