A Mutable Protocol for Consensus in Large Groups

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

R. Oliveira
Univ. 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 with a large number of participants. First, an apparently inefficient consensus protocol is developed using the very simple *stubborn channel* abstraction for unreliable message passing. Then, the introduction of judiciously chosen finite delays in the implementation of channels makes it likely that the transmission of some messages is avoided. Although this does not affect correctness, which rests on an asynchronous system model, the message exchange pattern at the network level changes noticeably and can be made to resemble several different protocols. A particularly appealing instantiation, called the *permutation gossip*, allows the protocol to scale gracefully to a large number of processes.**

I. Introduction

Several distributed programming problems such as atomic broadcast, view synchrony and atomic commitment can be reduced to consensus [1], 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 [2]. This impossibility can be circumvented by strengthening the asynchronous model with additional assumptions, and recently it has been shown that a large class of consensus protocols using different additional assumptions can be derived from the same generic framework [3].

We focus on protocols based on unreliable failure detectors [4], [5] in asynchronous message passing systems where processes 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 (average) number of communication steps (*i.e.* latency) and the number of messages in the network (*i.e.* bandwidth). First, we propose a new consensus protocol based on *stubborn channels* [7]. Al thought 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 of stubborn channels which maximizes the likelihood of desirable runs. Interestingly, this can be achieved simply by introducing finite delays in a naive implementation of stubborn channels. Such delays avoid actual transmission of messages because they increase the likelihood of a more recent message being sent in the meantime, which in stubborn channels discards all previously sent messages. 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. Different configurations of delays can also achieve different classes of desirable runs which resemble the message exchange pattern of different (well known or innovative) protocols.

Partially supported by FCT SHIFT (POSI/CHS/32869/2000).

As the proposed protocol can be made to behave at the network level like several different protocols we 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 presents the consensus protocol based on stubborn channels. Section 3 introduces protocol mutations and evaluates their performance. Section 4 briefly discusses related work and concludes the paper.

II. Mutable Consensus Protocol

In this section we introduce the mutable consensus protocol. First, we briefly describe the system model assumed and the definition of stubborn channels.

A. 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 has access to a local monotonically increasing clock. 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 [2]. Processes are completely connected through a set of fair-lossy communication channels [8]. 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.

B. Definition of Stubborn Channels

A stubborn channel [7] connecting two processes p_i and p_j is an unreliable channel defined by a pair of primitives $\mathsf{Send}_{i,j}(m)$ and $\mathsf{Receive}_{i,j}(m)$, 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 send a message m to p_j and p_i indefinitely delays sending any further message to p_j, then p_j eventually receives m.

Process p_i:

Function Consensus(v_i):
1 $est_i \leftarrow v_i$; $r_i \leftarrow 1$;
2 **while** $true$ **do**
3 $ph_i \leftarrow 1$; $P_i \leftarrow \emptyset$;
4 **if** $i = (r_i \mod n)$ **then**
5 $P_i \leftarrow \{i\}$;
6 **forall** k: $\mathsf{Send}_{i,k}((r_i, ph_i, P_i, est_i))$;
7 **endif**;
8 **while** $\#P_i \leq n/2$ **do**
9 **select**
10 **upon** $\mathsf{Receive}_{i,j}((r_j, ph_j, P_j, est_j))$:
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** $(r_i \mod n) \in P_i$ **then**
22 $est_i = est_j$;
23 **endif**;
24 **forall** k: $\mathsf{Send}_{i,k}((r_i, ph_i, P_i, est_i))$;
25 **endif**;
26 **upon** $\mathsf{Suspected}_i(j)$:
27 **if** $j = (r_i \mod n) \wedge ph_i = 1$ **then**
28 $ph_i \leftarrow 2$; $P_i \leftarrow \{i\}$;
29 **forall** k: $\mathsf{Send}_{i,k}((r_i, ph_i, P_i, est_i))$;
30 **endif**;
31 **endselect**;
32 **endwhile**;
33 **if** $ph_i = 1$ **then** return est_i; **endif**
34 $r_i \leftarrow r_i + 1$;
35 **endwhile**

Fig. 1. Mutable consensus.

A stubborn channel is easily implementable over a fair-lossy channel: It suffices to buffer the last message sent and periodically retransmit it.

C. Algorithm

In Fig. 1 we present an algorithm based on stubborn channels to solve the consensus problem [4]: 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: if a process decides v, then v was proposed by some process; no two processes decide differently; and every correct process eventually decides some value.

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.

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.

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.

D. Correctness Argument

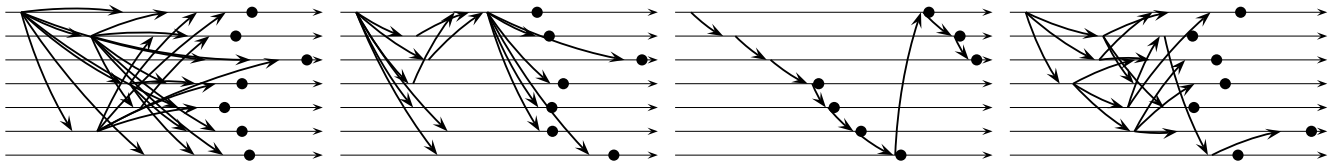
Due to lack of space, we omit a correctness proof [9]. Nevertheless, consider the following argument that the algorithm ensures validity and agreement. If a decision on v is reached in some round r, then 1) v is the estimate value est of the coordinator of r, and 2) any process p_i reaching a round $r' > r$ 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 collected 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 and confirming clause 2).

III. Protocol Mutations

The consensus protocol is now combined with various implementations of stubborn channels and evaluated. We start by introducing the experimental setting, then the implementations of stubborn channels, and finally the performance results obtained.



(a) Early (229 μs) (b) Centralized (447 μs) (c) Ring (754 μs) (d) Perm. gossip (428 μs)

Fig. 2. Prefixes of typical executions.

A. Experimental Setting

Common metrics often misrepresent the performance of distributed algorithms in complex environments such as the Internet [10]. Therefore we use a centralized simulation model [11] 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 [11] 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 implementation of the mutable consensus protocol used for evaluation is based on Java and run using Sun Java2 SDK with the HotSpot JIT compiler and Linux 2.4.21 operating system on a Pentium III 1GHz processor. Round and phase numbers are represented by 32 bit integers. The set of voters P_i is represented as a bitmap, making it compact for transmission and reducing set union to a bitwise or operation. The simulated application works as follows: Values are proposed simultaneously by all processes. When all processes decide, the system is restarted thus initiating a new run of consensus. The results presented in this paper are obtained without process crashes or suspicions.

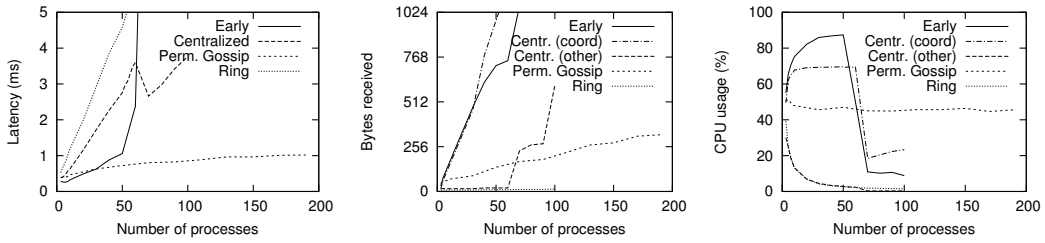
The centralized simulation runtime is also based on Java and uses a virtualized per-process CPU cycle counter to measure time. The simulated network used in this paper mimics a switched 10Mbps Ethernet network (*i.e.*, star topology). The model was calibrated by comparing results of benchmark applications with their counterparts running in a real system with identical characteristics. The model used does not however simulate scheduling latency, thus providing results that could only be observed in a real system if no other tasks were competing for the CPU or if a higher priority was assigned to the protocol task.

B. Implementation of Stubborn Channels

The implementation of stubborn channels assuming that an unreliable datagram service such as UDP/IP provides an implementation of a fair-lossy channel is straightforward. The sender keeps a buffer that can store a single message. The $\mathsf{Send}_{i,j}(m)$ primitive works as follows: Each message sent is buffered unaltered, discarding the previous message sent (if any). Periodically, the content of the buffer (if any) is transmitted using the datagram send primitive. The $\mathsf{Receive}_{j,i}(m)$ primitive is directly implemented by datagram receive.

Different implementations can be derived from this by introducing arbitrary finite delays. In detail, we consider the delays before the first actual transmission of the buffered message and between successive retransmissions. As long as these delays are finite, the implementation of the asynchronous specification of stubborn channels remains correct. To obtain the best performance we can therefore make use of whatever local knowledge is useful when computing delays, including the content of messages themselves.

In this paper, we propose four different strategies to compute such delays called *early*, *centralized*, *ring* and *permutation gossip*. A first intuition on the impact of such delays can be obtained from Fig. 2, which presents the graphical representation of actual runs of the mutable consensus protocol when combined with each of the implementations of channels. We call each of these combinations a *protocol mutation*. 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, from proposal to the last process deciding, is indicated in the caption. For clarity, only messages causally preceding decisions are presented. When using consensus as a building block in a distributed systems



(a) Latency (b) Network usage (c) CPU usage

Fig. 3. Performance of protocol mutations.

one is able to terminate the consensus instance after decisions and thus these are the relevant messages. Nevertheless, statistics in the next section account for all messages transmitted.

The *early* implementation of Fig. 2(a) is the simplest. It delays the initial transmission of messages when (i) the buffer was previously non-empty; (ii) the round/phase of the newly arrived message are the same as in the previous and (iii) the new message carries a minority of votes. 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. The result is 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.

The *centralized* implementation aims at producing a message exchange pattern which resembles that of the Chandra-Toueg centralized algorithm [4]. In detail, if the sender process is the current coordinator, the same delays of the *early* implementation are used. If not, messages are delayed only when the destination is not the current round coordinator thus making it likely that a decision is received from the coordinator and thus that the message is never actually sent. The impact is clearly visible in Fig. 2(b), showing the decisions occurring in three communication steps.

One can also achieve innovative message exchange patterns with desirable performance characteristics. The *ring* implementation delays messages from a process i to a process j unless $j = (i + 1) \mod n$ long enough for n message transmissions. The result is also obvious from the trace presented in Fig. 2(c) and shows a ring-style message exchange pattern in which each process communicates only with its successor.

Finally, the *permutation gossip* works as follows. A parameter F, called the fanout, is chosen. When a consensus instance is started, a random permutation of the set of process identifiers is generated by each process and used as circular list. An index into this list is initialized. For each message sent, if the destination is not within the next F processes in the list, the message is delayed. The index is then incremented by F. This results in a message exchange pattern that has the same desirable characteristics of probabilistic protocols while at the same time providing a correct (and deterministic) implementation of stubborn channels. Fig. 2(d) presents a run with $F = 3$.

C. Performance Evaluation

Although individual runs presented in Fig. 2 provide an intuition on the behavior of the protocol, the overall performance is better evaluated by statistics on protocol latency and resource usage. Fig. 3(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 sudden increase of latency of *early* and *centralized* mutations is explained by Fig. 3(b), which shows the average number of bytes received by a process during a run of consensus. 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 very large number of processes.

The effect of network congestion is also visible in Fig. 3(c), 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 become 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 don't scale regarding network and CPU usage. The *early* mutation is however still the best choice for small groups (*e.g.* less than 10 processes). Interestingly, almost all protocols proposed so far [4], [5], [12], [13], [3] 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 and low throughput. 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 and high throughput. Such message exchange pattern is also highly resilient to process failure and network omissions [9].

IV. Discussion

The proposal of generalized consensus protocols has been done before, namely regarding also the communication pattern [13] and to the oracle used [3]. The first approach [13] 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 [3] addresses only the issue of which oracle to use. This is orthogonal to our proposal and it should be possible to combine them.

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 tuning procedure operates only in the time domain and thus does not, in any way, affect the protocol which assumes an asynchronous system model. This has some interesting consequences. First, it fosters innovation by making it safe to experiment with innovative message exchange patterns. In addition, in contrast to protocol layer switching [14] 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. This means also that, given an adequate policy, it is trivial do dynamically reconfigure the protocol to adapt to a changing environment.

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 a semantically reliable multicast, which generalizes the stubborn channel abstraction to multicast communication [15]. 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 the prototype of GROUPZ, a group communication toolkit based on mutable protocols which is configured by selecting implementations of stubborn channels.

References

[1] R. Guerraoui and A. Schiper, "The generic consensus service," *IEEE Trans. Software Engineering*, vol. 27, no. 1, Jan. 2001.

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

[3] R. Guerraoui and M. Raynal, "The information structure of indulgent consensus," Tech. Rep.PI-1531, IRISA, Apr. 2003.

[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, Dép. 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 *Intl. Ws. Distributed Algorithms*, Oct. 1996.

[9] J. Pereira and R. Oliveira, "The mutable consensus protocol," Tech. Rep., Dep. de Informática, University of Minho, July 2003.

[10] I. Keidar, "Challenges in evaluating distributed algorithms," vol., 2584 of *Lecture Notes in Computer Science*. Springer, 2003.

[11] 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.

[12] R. Oliveira, *Solving Consensus: From Fair-Lossy Channels to Crash-Recovery of Processes*, Ph.D. thesis, Dép. d'Informatique, École Polytechnique Fédérale de Lausanne, Feb. 2000.

[13] M. Hurfin and A. Mostefaoui and M. Raynal, "A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors," *IEEE Trans. Computers*, vol. 51, no. 4, Apr. 2002.

[14] 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.

[15] Pereira, J. and Rodrigues, L. and Oliveira, R., "Semantically reliable multicast: Definition, implementation and performance evaluation," *IEEE Trans. Computers*, vol. 52, no. 2, Feb. 2003.