Randomized Consensus in Wireless Environments: A Case Where More is Better

Bruno Vavala, Nuno Neves, Henrique Moniz, Paulo Veríssimo
LaSIGE, University of Lisbon - Portugal
{*vavala, nuno, hmoniz, pjv*}*@di.fc.ul.pt*

***Abstract*—In many emerging wireless scenarios, consensus among nodes represents an important task that must be accomplished in a timely and dependable manner. However, the sharing of the radio medium and the typical communication failures of such environments may seriously hinder this operation. In the paper, we perform a practical evaluation of an existing randomized consensus protocol that is resilient to message collisions and omissions. Then, we provide and analyze an extension to the protocol that adds an extra message exchange phase. In spite of the added time complexity, the experiments confirm that our extension and some other implementation heuristics non-trivially boost the speed to reach consensus. Furthermore, we show that the speed-up holds also under particularly bad network conditions. As a consequence, our contribution turns out to be a viable and energy-efficient alternative for critical applications.**

I. Introduction

Consensus is a generic abstraction for activity coordination in distributed environments, where nodes propose some local value and then they all reach the same result. In several emerging wireless scenario, the nodes' capability of performing coordination tasks is of significant growing interest due to its disparate practical applications. Car platooning in vehicular networks and computing with swarms of agents are just few examples. The first is aimed at grouping cars and making them agree on a common speed, in order to improve highway throughput. The second instead enables to take advantage of the collective behaviour of a self-coordinated set of agents. Its usage spreads from the control of unmanned vehicles to sensor monitoring and actuation. In these settings, since the presence of faults can neither be disregarded nor prevented, it becomes necessary to tolerate them using appropriate protocols.

Fault-tolerant consensus protocols have been matter of research for decades. Proposals have been made for a range of timing models, from synchronous to asynchronous. The asynchronous model allows for the most generic implementations as it avoids any sort of timing assumptions, increasing the resilience to unplanned delays (e.g., because of node or network overloads). However, it is bound by an impossibility result that prevents the deterministic solution of consensus in presence of one faulty node (FLP result) [1]. In any case, even increasing the assumed synchrony is not a panacea, if the communication among nodes is not reliable. Under the *dynamic omission failure model*, a majority of nodes cannot deterministically reach consensus if $n-1$ omission faults can occur per communication step, in a synchronous system with n nodes (SW result) [2]. Due to this restrictive result, this model has not been used often, even though it captures well the kind of failures that are observed in wireless ad hoc networks. For instance, dynamic and transient faults caused by environmental conditions, and the temporary disconnection of a node.

Over the years, several extensions to the asynchronous model have been proposed to evade the FLP result: randomization is one of these [3]. Only recently, has this same technique been successfully applied to circumvent the SW impossibility result [4]. Randomization however has always been considered a significant theoretical achievement, but much less a practical one. According to many theoretical studies, randomized consensus protocols are inefficient because of their expected high time and message complexities.

In this paper, we argue and provide evidence that it is possible to build randomized protocols smartly, so that they represent a feasible and practical alternative in wireless environments. Firstly, we analyze the performance of the randomized protocol [4], which has been built for the dynamic omissions failure model. Currently, there is a lack of experience on the implementation and evaluation of protocols for this model. The selected protocol has some nice characteristics, such as ensuring *safety* despite of an unrestricted number of omission faults, and *liveness* when the number of such faults is less than some bound. Secondly, we propose an extension to the protocol, in particular the addition of an extra message exchange phase (a third phase). Results show that even though our new solution slightly worsens the best case scenario, it allows significant improvements in all the other cases, even in presence of bad network conditions. The reached speed-up translates not only into lower latencies, but also into less broadcasts, less network usage, thereby enabling our extended protocol to be practical for both time and energy critical environments.

II. System Model

The system is composed by a set of n processes with identities $\mathcal{P} = \{p_0, p_1, \ldots, p_{n-1}\}$. It is completely asynchronous in the sense that there is no upper bound on the delays to deliver a message and on the relative speeds of processes. Since our aim is to provide a protocol for wireless networks, the communication medium is shared among processes and every transmission turns out to be a message broadcast. It is assumed that processes are within range of each other. We consider the dynamic communication failure model [2], which captures well the nature of communication problems that may occur, such as dynamic and transient message omissions. It does not make any assumption about the fault patterns, it only presupposes that such faults last for a finite period of time. What may happen is that a process omits a message broadcast or fails to receive a message. The first case might be due to a process that crashes or is temporarily disconnected, and the second case can be related to collisions or environmental noise.

III. The Consensus Problem

The *k-consensus* problem considers a set of n processes where each process p_i proposes a binary value $v_i \in \{0, 1\}$, and at least $k > \frac{n}{2}$ of them have to decide on a common value proposed by one of the processes. The remaining $n-k$ processes do not necessarily have to decide, but if they do, they are not allowed to decide on a different value. Our problem formulation is designed to accommodate a randomized solution and is formally defined by the properties:

- Validity: If all processes propose the same value v, then any process that decides, decides v.
- Agreement: No two processes decide differently.
- Termination: At least k processes eventually decide with probability 1.

IV. The Randomized Consensus Protocol

The paper studies two versions of a consensus protocol (see Algorithm). The first corresponds to the protocol of [4], whose authors proved the correctness but did not provide an experimental evaluation. This protocol was originally built for a synchronous environment, but with the right receive primitive it can also be used in an asynchronous setting. The second version is an extension that incorporates a third phase (darker box in Algorithm). We do not provide here its correctness proof due to lack of space, but a simple comparison between the two algorithms shows that it can be easily adapted from the former.

A. Overall Execution

Computation proceeds in asynchronous rounds. In each one of these, a process performs a message broadcast of its status (line 6), and after that, it invokes *smart-receive()* to get some messages (line 7). Then, based on the collected messages, it may perform some local computation (lines 9-36). In our context, *smart-receive()* can have different implementations that do not compromise correctness and will be pointed out later.

The status of a process p_i defines the current configuration and it is composed by a set of internal variables: the phase number $\phi_i \geq 0$ (initially set to 0); the proposal $v_i \in \{0, 1\}$ (initially set to the proposal provided as parameter); the decision status $status_i$ (initially is $undecided$).

```
   Input: Initial binary proposal value $proposal_i \in \{0,1\}$
   Output: Binary decision value $decision_i \in \{0,1\}$
1  $\phi_i \leftarrow 0$;
2  $v_i \leftarrow proposal_i$;
3  $status_i \leftarrow undecided$;
4  $V_i \leftarrow \emptyset$;

5  while true do
6      broadcast($m_i := \langle i, \phi_i, v_i, status_i \rangle$);
7      $\boxed{\mathcal{M} \leftarrow \text{SMART-RECEIVE}(timeout);}$
8      $V_i \leftarrow V_i \cup \mathcal{M}$;

9      if $\exists \langle *, \phi, v, status \rangle \in V_i : \phi > \phi_i$ then
10         $\phi_i \leftarrow \phi$;
11         $v_i \leftarrow v$;
12         $status_i \leftarrow status$;
13     end

14     if $|\{m \in \{\langle *, \phi_i, *, * \rangle\} \subseteq V_i\}| > \frac{n}{2}$ then
           if $\phi_i \bmod 3 = 0$ then
               $v_i \leftarrow_{v \in \{0,1\}}^{\max} |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}|$;
           else
15
16         if $\phi_i \bmod 3 = 1$ then
17             if $\exists\, v \in \{0,1\} : |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}| > \frac{n}{2}$
               then
18                 $v_i \leftarrow v$;
19             else
20                 $v_i \leftarrow \perp$;
21             end
22         else
23             if $\exists\, v \in \{0,1\} : |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}| > \frac{n}{2}$
               then
24                 $status_i \leftarrow decided$;
25             end
26             if $\exists\, v \in \{0,1\} : |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}| \geq 1$
               then
27                 $v_i \leftarrow v$;
28             else
29                 $v_i \leftarrow \texttt{coin}_i\texttt{()}$;
30             end
31         end
32         $\phi_i \leftarrow \phi_i + 1$;
33     end
34     if $status_i = decided$ then
35         $decision_i \leftarrow v_i$;
36     end
37 end
```

Algorithm: The 3-Phase Consensus Protocol.

In the protocol execution, there is a difference between the concepts of *round* and *phase*. A round corresponds to a full iteration of the $while$ loop, starting from line 5 and ending in line 37. A phase is implemented as a process' local variable (ϕ_i), whose value increases monotonically as enough good messages are received, namely when a process is able to update its status (line 32). Processes do not necessarily have the same phase while they execute consensus concurrently. A process may be temporarily outside the communication range of the others, thereby being unable to make progress and increase the phase number (but continues to execute the loop). However, due to the transitory nature of such situation, as soon as it is able to receive messages, possibly carrying a phase higher than its, it can catch up immediately with the other processes by updating the status (lines 9-13).

After broadcasting the status, the process blocks in *smart-receive* to obtain some messages (line 7). This function returns a set $\mathcal{M}$ of messages, all of which are stored in a vector V_i (line 8), if not yet received (this is implied by the union which avoids storing duplicates). More than $\frac{n}{2}$ of these are needed to pass successfully through the first *if* and make progress (line 14). Indeed, after the first *if*, no matter what happens next, the process updates its status at least by increasing the phase (line 32).

Now let us assume that enough messages have been received. The process executes instructions in the black box (the extra phase) because the current phase number is $\phi_i = 0$. This preliminary phase was added to help processes converge rapidly to a decision. Basically, a process sets the local proposal value to a (weak) majority of the proposals carried in the messages (if there is a tie, the process selects value 0). We will show that if processes start with divergent proposals (some of them with 0 and others with 1), this additional step makes most (possibly all) of them choose equal values before moving to the next phase (line 32).

After receiving enough messages, the process executes the second phase (lines 16-22) because $\phi_i = 1$. Here, if more than $\frac{n}{2}$ messages carry the same proposal value v, then the process updates the local proposal to this majority value (line 18). Otherwise, the process chooses the default value $\perp \notin \{0, 1\}$ (line 20). One should note that this procedure ensures that if any other process p_k sets $v_k \in \{0, 1\}$, then v_k will be equal to v_i because of the strong majority imposed by $\frac{n}{2}$ (line 17). Before moving to the next round, the process increases the phase number (line 32).

In the third phase, $\phi_i \bmod 3 = 2$, the process tries to make a decision (lines 22-31). If it receives more than $\frac{n}{2}$ messages with the same value (different from $\perp$), then it is allowed to decide on that value (line 24, 27 and then lines 34-36). Moreover, there is the guarantee that if any other process decides, it will do it for the same value because of the imposed strong majority of $\frac{n}{2}$. If all messages carry as proposal $\perp$, meaning that no process has a preference for a decision value, then the process sets the proposal to an unbiased coin that returns 0 or 1 with equal probabilities (line 29). Eventually, after some rounds, with probability 1 enough processes will get the same coin value that will allow the protocol to make a decision.

B. Receive Operation

A process p_i blocks in the *smart-receive()* operation to collect messages in order to make progress in the protocol execution (either by entering the *if* in line 9 or 14). However, as there may be omission failures, the process does not know how many messages may arrive in a given round, and therefore a timeout mechanism must be implemented inside the *smart-receive()*. When the timeout expires, even if not enough messages have been delivered, the operation is required to return (the reader should note that the use of this timeout does not violate our asynchronous assumption, as it is local and could be implemented with a simple counter). This allows the process to initiate a new round, where the status message is retransmitted (line 6), and then p_i can wait for the reception of a few more messages (line 7). Since this procedure is carried out by all processes, eventually p_i will get sufficient messages to advance.

Consequently, a pondered implementation of this operation is fundamental to achieve good performance because it defines the instants when progress can be made and how often messages are broadcast. We adopted and evaluated two strategies in our current implementation.

In the first strategy, the operation waits for the arrival of $\lfloor \frac{n}{2} + 1 \rfloor$ different messages with the same phase of the process (or for the timeout to expire). As soon as this amount is received, the function immediately returns. Other messages received with that phase (in the next rounds) will be considered old and discarded. To simplify the calculations, we always set the timeout to $10ms$. We call this option *Immediate Progress* (ip).

In the second strategy, the operation waits for all messages that may arrive in a timeout period, possibly much more than $\frac{n}{2}$. After the timeout expires, the operation returns this whole set of messages. Here the timeout value is more critical as we want to wait for a set of messages, such that $\lfloor \frac{n}{2} + 1 \rfloor \leq |\mathcal{M}| \leq n$, but without wasting too much time if messages get lost. In our experimental environment, it was found that $timeout = n \times 1.25\ ms$ provides good results. It is a matter of future research to devise a mechanism to adapt the timeout to the current network conditions. We call this option *No Immediate Progress* (no-ip).

C. Protocol Termination

It is worth to notice that the protocol guarantees termination, in the sense that consensus is eventually reached, but it does not stop the execution. Here the problem is that when a process decides, it must keep on broadcasting its status to let the others decide. Furthermore, even if it learns that everyone has decided, the process cannot be allowed to terminate because someone may have not received its decision status and would never terminate.

One solution to this problem is to use a centralized node that is informed when any process decides, and then tells everyone that they should terminate when enough processes have finished. This approach has several limitations, such as how to address the failure of the centralized node. In our implementation, we resorted to a pragmatic solution that is based on giving *sufficient* time after decision for the processes to terminate. In more detail: a process continues to broadcast up to a certain time after decision (1 second); then, the process is allowed only to receive messages, to empty its network buffer; if no message is received for some interval (2 seconds), then the process terminates. Clearly it is not a perfect solution, but it worked very well in our experimental setting. Additionally, it did not impact the evaluations because the utilized metric was the latency (defined in the next section).

V. Performance Evaluation

In this section we evaluate the two versions of the protocol, stressing how little modifications may produce so different and perhaps unexpected results.

A. Testbed and Implementation

The experiments were carried out in the Emulab testbed [5]. All the nodes, located few meters away from each other, were equipped with a 600 MHz Pentium III processor, 256 MB of RAM, and the 802.11g D-Link DWL-AG530 WLAN interface card. The nodes run Fedora Core 4 Linux, with the 2.6.18.6 kernel version.

The protocol versions were implemented in C, and they used UDP for lower level communication to take advantage of the broadcast medium. During the evaluations, we observed some omission failures, which in part were due to collisions caused by our own traffic (the testbed is shared with other researchers, and therefore, their experiments also created collisions). Nevertheless, in order to evaluate the protocols in presence of omissions, we decided to develop a layer to emulate network losses. Such layer represents our *adversary*. The adversary is able to delete a complete message broadcast and to prevent specific processes from successfully receiving a message. The decision of which messages should be discarded is made randomly. The user can specify different percentages of messages to be removed at the source and at the destination.

The main metric we use to compare the different approaches is *latency*. The *latency at process* p_i is defined as the interval between the moment the protocol starts and the instant when the local decision is reached. The *latency of an experiment* is the average value of the latencies of all processes. In the graphs, we present average values of several experiments to have a good confidence on the results.

B. Experimental Results

1) Uniform proposal distribution: We evaluate the best scenario in which all processes start with the same (uniform) proposal value. As by the consensus specification (see Section III), this value is the only one that can be decided. The two protocol versions reach a decision after 2 phases in the original version, and after 3 phases in our version. In the experiments, we observed that processes always receive enough messages to make progress in every round, allowing termination to be reached in these minimum number of phases (and with the minimum number of rounds). *Fig. 1* shows clearly the increase in delay that the new phase

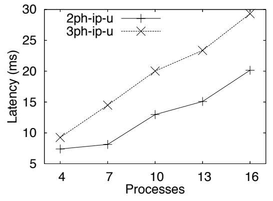

Figure 1: *Latency with uniform proposal values.*

adds. The slopes of the lines reflect the difference between executions that require respectively $2n$ and $3n$ broadcasts.

2) Divergent proposal distribution: In the rest of the experiments, we will always consider a divergent initial proposal distribution among processes (half of them propose 0 while the others propose 1). Before highlighting the difference in execution speeds between the protocol versions, we supply some results related to the introduction of the *no immediate progress* (*no-ip*) concept in the *smart-receive()* operation. *Fig. 2* shows that the concept is important in enhancing performance (smaller latency is better).

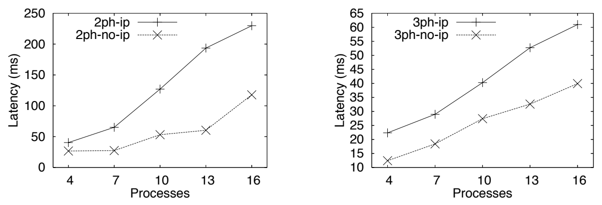

Figure 2: *Evaluation of the* smart-receive() *strategies.*

In a context where initial proposals differ from each other, even though the *ip* strategy allows a process to make progress as soon as possible, this will (very probably) lead to nowhere. In order to make progress towards a decision, a process needs more than $\frac{n}{2}$ messages with the *same value* to avoid default values (line 20) and random choices (line 29). Therefore, with *ip*, the process needs to be lucky with the order of the messages it receives to converge to a decision. On the other hand, with *no-ip*, waiting for more messages allows one to exploit the power of random choice to bias the proposals toward a particular value.

Let us clarify this issue with an example in a fault-free scenario. Suppose that n is even, $n \geq 4$ and processes have *divergent* proposals. Since there is no strong majority, all of them will be compelled to toss a coin (with or without *immediate progress*) after running the first phases (line 29). Now, let $V[i]$, $0 \leq i \leq n-1$ be a vector with the new proposals and consider the event $\mathcal{E} := \{\sum_{i=0}^{n-1} V[i] \ equals \ \sum_{i=0}^{n-1} 1 - V[i]\}$. After one coin tossing we have that $P(\neg\mathcal{E}) > P(\mathcal{E})$, hence: (1) proposals tend to be equally distributed between the values, but (2) the chances to get a strong majority overtake the others (more and more as n increases). Therefore, with *no-ip* processes will probably progress toward consensus because of (2), while with *ip* it would be very difficult for processes to notice a strong majority (among only $\lfloor \frac{n}{2} + 1 \rfloor$ messages received) in the first phase because of (1).

3) The addition of the third phase: The addition of an extra phase has potentially several drawbacks. Namely, it increases the network usage because consensus can only be reached in phases that are multiple of 3 rather than 2. The results in *Fig. 3* show instead that the 3-phase protocol overwhelms the 2-phase one in both strategies.

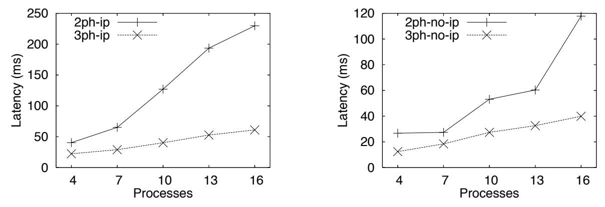

Figure 3: *The impact of the third phase.*

Actually, the third phase represents a smart algorithm design that at first might escape our (theoretical) analysis. Without this phase, a process needs to receive more than $\frac{n}{2}$ messages with the same value to set its proposal (line 18) and eventually decide (line 24). However, according to all possible configurations, it is unlikely that such strong majority occurs, even more with a divergent proposal distribution. This situation forces a lot of processes to set default (line 20) and random values (line 29), preventing them from deciding and making them start all over again. The problem here is that messages end up being discarded too easily and with little consideration, without exploring as much as possible the information being transmitted.

This is what the third phase does – it uses better the data carried in the messages, to allow progress towards a decision much sooner. In the original protocol, the first *prepare phase* was intended to make the processes set the same value (lines 16-22), while the second *decision phase* was to let them learn and decide on this value (lines 22-31). Our third phase is executed just before these two (resulting in an extra *pre-prepare phase*), and it relaxes the need of a strong majority. Altough more than $\frac{n}{2}$ messages are always needed to make progress, here there is no other requirement: a process sets its proposal to the majority value received (and in case of a tie selects value 0). Clearly, it is not true that the processes will always set their proposal to the same value, since this depends on the ordering of message arrival. Nevertheless, since processes share the same wireless medium, reordering is expected to be small and all processes should receive approximately the same set of messages. Hence, if there is a value that is predominant in the set, this enables lots of processes (if not all of them, in expectation) to pre-set that value. Therefore, this has a positive influence on the subsequent phases, improving convergence. Indeed, our experiments show that most of the times consensus is reached after 3 phases, and rarely in more than 6.

4) Performance under failure scenarios: The three-phase version outperforms the original protocol in presence of an adversary that creates various sorts of omission failures. Such adversary may make a process discard an entire broadcast (causing n receive events to be lost), with probability $\mathcal{P}^s$, or may cause a single message omission at a specific receiver, with probability $\mathcal{P}^r$. The probabilities that were used in the experiments are: $\mathcal{P}^s_\alpha = 0.1$ and $\mathcal{P}^r_\alpha = 0.3$, abbreviated as adversary *adv-1-3*; and $\mathcal{P}^s_\beta = 0.3$ and $\mathcal{P}^r_\beta = 0.6$, abbreviated as adversary *adv-3-6*. According to these probability, defining the event $\mathcal{E} := \{\text{message reaches destination}\}$, $\mathcal{P}_\alpha[\mathcal{E}] = (1 - \mathcal{P}^s_\alpha)(1 - \mathcal{P}^r_\alpha) = 0.63$ and $\mathcal{P}_\beta[\mathcal{E}] = (1 - \mathcal{P}^s_\beta)(1 - \mathcal{P}^r_\beta) = 0.28$. Adding up to these failures, we should not forget that there is already some packet loss due to wireless links and the (unreliable) UDP protocol. The experiments show that these omissions are almost null with a few processes but they can increase up to 30% of the traffic with 16 nodes. The results with such severe conditions are visible in *Fig. 4* and *Fig. 5*.

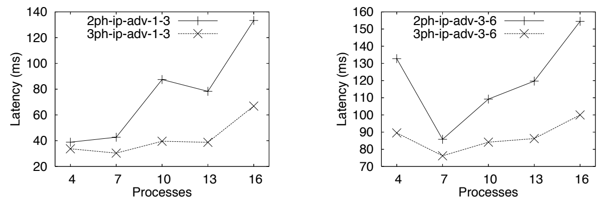

Figure 4: Immediate Progress *in presence of an adversary.*

In the figures it is visible a curious latency trend, where often it decreases when the number of nodes is raised from 4 to 7 and from 10 to 13. This is due to the majority threshold of $\lfloor \frac{n}{2} + 1 \rfloor$. While the number of processes is always increased by 3 units, the majority threshold increases less regularly. More specifically, it increases in 1 unit in those previous cases, and in 2 units in the others. Furthermore, the ratio between majority and number of processes turns out to decrease in those cases and to increase in the others.

It is also possible to notice that the 4 nodes configuration in the *2ph-ip-adv-3-6* experiment has particularly high latency. This setting is very sensitive to packet dropping, and therefore, nodes need to perform more broadcasts, most of which are duplicates (retransmissions of their status).

It is noteworthy to consider something unexpected – the adversary *adv-1-3* can make the 2-phase protocol with *immediate progress* go faster (compare *Fig. 3* with *Fig. 4*). This is not (only) due to the entire broadcasts that are discarded, thereby reducing wireless medium contention, and therefore improving message delivery. According to the test data retrieved in those cases: less phases and less broadcasts are needed to reach consensus; less status information is sent and received for every phase; more phase jumps occur due to the arrival of messages with higher phase; few processes reach decision by themselves. Therefore, what happens is that, in every phase, fewer processes are able to make progress and, as soon as they update their status and broadcast it, other processes that were left behind can immediately catch up with them, learning by copying their status (lines 9-13). It is this copy that boosts the decision procedure because processes use it as a sort of *pre-prepare phase*, avoiding setting default and random values.

In any case, though more onerous in theory, our phase

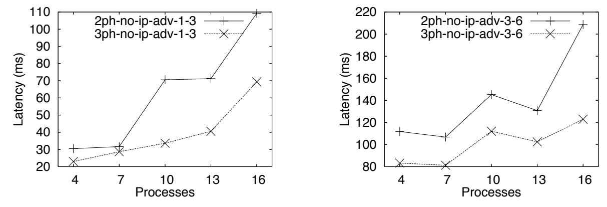

Figure 5: No Immediate Progress *in presence of an adversary.*

extension enables the processes to complete the task even more quickly in all the cases and, above all, provides results much more stable, less subject to relevant changes due to the lower number of coin flips induced.

VI. Related work

Consensus plays a pivotal role in distributed computing, particularly when a system needs to cope with accidental faults (e.g., node crashes). The use of randomization in this context arose due to the necessity to circumvent the well know FLP impossibility result [1]. The first seminal works that used this technique were due to Ben-Or [6] and Rabin [7]. Both of them provided protocols to deal with arbitrary node faults, and they run in an expected exponential number of rounds. Later, Bracha [8] published an optimal protocol to cope with fail-stop processes based on a local coin paradigm. Cachin et al. [9] presented the ABBA protocol for Byzantine agreement, resorting to a shared coin paradigm and asymmetric cryptography operations. A more detailed survey on this class of protocols is available in [3].

To the best of our knowledge, research in randomized protocols has been mostly theoretical, probably because of their exponential complexity. Moniz et al. [10] made a detailed performance comparison between ABBA (for the shared coin class) and Bracha (for the local coin class) protocols. According to their results, the local coin protocol outperformed the shared coin protocol when there is high availability of network bandwidth, which is typical in a LAN. When the bandwidth starts to lower and the communication delays increase, as in WANs, the cost of cryptographic operations becomes less important and shared coin protocols can take advantage of their constant expected running time. The same authors also did an evaluation of a speed agreement algorithm in the context of car platooning, using a stack of intrusion-tolerant protocols [11].

In this paper, we study a different type of randomized protocol [4], which was designed to work under the dynamic omissions failure model (and also to circumvent the SW impossibility result [2]). We also propose a set extensions, including a third phase. Contrarily to intuition, since the extra phase increases the overall time complexity, the new version of the protocol much more efficient in practice.

VII. Conclusions

The paper provides evidence of the practicality of randomized consensus protocols. Even though such protocols have a high theoretical complexity, the experiments show that performance can be significant even with high failure rates. This is particularly true in realistic scenarios where a small number of process is being used and if algorithms are carefully engineered to be as efficient as possible. A protocol previously present in the literature, which aimed at solving consensus in wireless environments, has been analyzed and extended. We show how these extensions can exploit the available information in order to increase performance. The result is a protocol that is much faster in terms of latency, more thrifty in terms of messages and hence of network usage, properties that make if effective for time and energy constrained environments. In the future we plan to carry out extensive comparisons with other consensus protocols under the unreliable network communication model.

Acknowledgments

This work was partially supported by the FCT through the Multiannual and the CMU-Portugal Programmes, and the project PTDC/EIA-EIA/100894/2008 (DIVERSE).

References

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[2] N. Santoro and P. Widmeyer, “Time is not a healer,” in *Proc. of the 6th STACS Symposium*, 1989, pp. 304–313.

[3] J. Aspnes, “Randomized protocols for asynchronous consensus,” *Distrib. Comput.*, vol. 16, no. 2-3, pp. 165–175, 2003.

[4] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo, “Randomization can be a healer: Consensus with dynamic omission failures,” in *Proc. of the 23rd DISC Symposium*, 2009, pp. 63–77.

[5] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An integrated experimental environment for distributed systems and networks,” in *Proc. of the OSDI Symposium*, 2002, pp. 255–270.

[6] M. Ben-Or, “Another advantage of free choice: Completely asynchronous agreement protocols,” in *Proc. of the 2nd ACM PODC Symposium*, 1983, pp. 27–30.

[7] M. O. Rabin, “Randomized Byzantine generals,” in *Proc. of the 24th Annual IEEE FOCS Symposium*, 1983, pp. 403–409.

[8] G. Bracha, “An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol,” in *Proc. of the 3rd ACM PODC Symposium*, 1984, pp. 154–162.

[9] C. Cachin, K. Kursawe, and V. Shoup, “Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography,” *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[10] H. Moniz, N. Neves, M. Correia, and P. Veríssimo, “Experimental comparison of local and shared coin randomized consensus protocols,” in *Proc. of the 25th IEEE SRDS Symposium*, 2006, pp. 235–244.

[11] H. Moniz, N. F. Neves, M. Correia, A. Casimiro, and P. Verissimo, “Intrusion tolerance in wireless environments: An experimental evaluation,” in *Proc. of the 13th PRDC Symposium*. IEEE Computer Society, 2007, pp. 357–364.