Randomized Consensus in Wireless Environments

Bruno Vavala, Nuno Neves, Henrique Moniz, Paulo Veríssimo
Department of Computer Science
LaSIGE, University of Lisbon - Portugal
Email: {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 describe an interesting relationship with a totally different protocol, which explains why the speed-up holds and improves also under particularly bad network conditions. As a consequence, our contribution turns out to be a viable and energy-efficient alternative for critical applications.**

***Keywords*-randomized consensus; wireless networks; message omissions; asynchrony**

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 scenarios, the nodes' ability to perform coordination tasks is of growing interest due to various 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 enables a set of agents to self-coordinate to take advantage of the collective behavior, and 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) [2]. 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 more than $n-2$ omission faults can occur per communication step, in a synchronous system with n nodes (SW result) [3]. 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 [4]. Only recently, has this same technique been successfully applied to circumvent the SW impossibility result [5]. 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 [5], 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. In this last case, the algorithm is sometimes even faster than under normal network conditions. We explain such unexpected result by showing a connection between our protocol and a previously studied other form of consensus. In the end, 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.

The rest of this paper is organized as follows: in Section II we give a roadmap of the research work in the area; in Section III we describe the system model that underlies our algorithm; in Section IV we briefly detail the k-consensus problem; in Section V we present the extended algorithms, with a detailed explanation of each step; in Section VI we sketch the proof of correctness of the algorithms; in Section VII we supply the results of several experiments, justifying them; in Section VIII we shortly discuss our result and outline some directions for future research.

II. 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 [2]. 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, which terminate 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 the shared coin paradigm and asymmetric cryptography operations. A more detailed survey on this class of protocols is available in [4].

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 becomes smaller and the communication delays increase, as in WANs, the cost of cryptographic operations is less important and the 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]. Another interesting evaluation is conducted in [12]. A deterministic algorithm for atomic broadcast is tested under high load, in order to chase (without success) the FLP result.

In this paper, we study a different type of randomized protocol [5], which was designed to work under the dynamic omissions failure model. We also propose a set extensions, including a third phase which, contrarily to intuition, results in a new version of the protocol much more efficient in many practical circumstances. The omission failure model was proposed to show that a reliable asynchronous system is not so different from an unreliable synchronous one, bounded by the SW impossibility result [3]. Indeed, [13] provides a note on their equivalence by simulating one with the other. Other similar and extended impossibility results can be found in [14], [15], by reasoning about knowledge [16], or through a layered analysis of distributed systems [17]. Recently, the work in [5] has been further extended to accommodate Byzantine failures [18], thus closing the long-standing open gap in synchronous systems. However, as well as it happened in any other work that used randomization against a strong adversary to circumvent an impossibility result, the complexity of the proposed protocol in the worst-case scenario is exponential in the number of processes.

III. 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 [3], 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.

IV. 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.

V. The Randomized Consensus Protocol

The paper studies two versions of a consensus protocol (see Algorithm 1). The first corresponds to the protocol of [5], 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 1). We also provide a sketch of its correctness proof, showing how it can be easily adapted from the one of the former algorithm.

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 state 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 $undecided$).

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 state (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 to 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 state (lines 9-13).

In more detail, after broadcasting the state, 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 main *if* and make progress (line 14). Indeed, after the *if*, no matter what happens next, the process at least updates its state by increasing the phase (line 32).

Now, the process executes instructions in the black box, the extra phase that we call *Pre-Prepare 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 an equal value before moving to the next phase (line 32).

After receiving enough messages, the process executes the second phase (lines 16-22), that we call *Prepare Phase*, 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, $\phi_i \bmod 3 = 2$, called *Decision Phase*, 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 more than $\frac{n}{2}$ messages with the same value. 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 careful 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

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

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

    if $\exists \langle *, \phi, v, status \rangle \in V_i : \phi > \phi_i$ then                /* Catch-Up Block */
        $\phi_i \leftarrow \phi$;
        $v_i \leftarrow v$;
        $status_i \leftarrow status$;
    end

    if $|\{m \in \{\langle *, \phi_i, *, * \rangle\} \subseteq V_i\}| > \frac{n}{2}$ then
        if $\phi_i$ mod 3 = 0 then
            $v_i \leftarrow \max_{v \in \{0,1\}} |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}|$;   /* Pre-Prepare Phase $\phi_i \pmod 3 = 0$ */
        else

        if $\phi_i$ mod 3 = 1 then                                  /* Prepare Phase $\phi_i \pmod 3 = 1$ */
            if $\exists v \in \{0,1\} : |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}| > \frac{n}{2}$ then
                $v_i \leftarrow v$;
            else
                $v_i \leftarrow \perp$;
            end
        else                                                        /* Decision Phase $\phi_i \pmod 3 = 2$ */
            if $\exists v \in \{0,1\} : |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}| > \frac{n}{2}$ then
                $status_i \leftarrow decided$;
            end
            if $\exists v \in \{0,1\} : |\{m \in \{\langle *, \phi_i, v, * \rangle\} \subseteq V_i\}| \geq 1$ then
                $v_i \leftarrow v$;
            else
                $v_i \leftarrow$ coin$_i$();
            end

        $\phi_i \leftarrow \phi_i + 1$;
    end
    if $status_i = decided$ then
        $decision_i \leftarrow v_i$;
    end
end
```

Algorithm 1: The 3-Phase Consensus Protocol.

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, which we call *No Immediate Progress* (no-ip), 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 clear that having such static timeouts is not the best option, and things like network load should be taken into consideration. Therefore it would be useful to devise mechanisms to adapt the timeouts to the current network conditions. However, this topic falls outside the scope of this paper and will be matter of future investigation. For instance, the work of [19] could be used as a starting point, since it is proposed a pro-active method for checking the network quality of service and estimating the timeout accordingly.

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 (and cannot) 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 is not allowed to terminate because someone else may have not received its decision status and would never terminate. Again, even if everyone else had received it, no process would be sure that any other process knows that they all reached consensus. In summary, this is the classic problem that arises in any unreliable message-passing systems, and which is impossible to solve through a finite number of message exchanges, without further assumptions. More precisely, as it has been formally proved in [17], in such systems the processes are unable to attain the required level of knowledge (i.e. common knowledge) about consensus termination, to stop sending messages safely.

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 the consensus. Clearly it is a far from being 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). The same applies to the case when we count the number of rounds to reach consensus, because counting is stopped by that time.

VI. Correctness

As previously underlined, the algorithm is an extension of the one proposed in [5], hence its correctness follows almost immediately from it. For this reason, we only sketch the proofs for the sake of completeness.

Firstly, we give some straightforward hints on why the asynchronous model does not represent a problem and why the usage of timeouts does not contrast with it. By simply looking at the algorithm, it is clear that there is no time assumption, but the solely presence of a timeout. The computation of each process is thus message-driven, and not directed by clock-synchronization, in the sense that they need either a single message carrying a larger phase or more than $\frac{n}{2}$ of them with the same phase to set some value and go ahead. The algorithm however does not have any mechanism to detect failures (message dropping). Therefore, a process cannot determine if a message has not been delivered because it travels too slowly or because it was dropped. The use of a timeout is sufficient to solve this issue, by making the sender broadcast the same message infinitely often, if it is unable to hear anything from (enough of) the other processes. The timeout does not violate any asynchrony assumption because it does not necessitate any synchronization with global time. As stated in [20], it can be implemented locally, following just the process' local clock, by a simple (instruction) counter. The only drawback of this implementation is that the rebroadcast operation is static and timeout-driven, so in practice it could overload the network. However, there exist ongoing works, aimed at overcoming such inefficiency by dynamically linking the timeout to the network conditions (see [19]).

For what concerns the correctness proof, we divide it into two branches: *safety* and *liveness*. The safety part is straightforward because the algorithm has been designed to be always safe no matter what the time requirements, the number of omissions and their patterns are. So, starting from an initial safe state, either it does not make any progress (it does not receive enough messages for the computation to evolve to higher phases), thus remaining in a safe state, or it does, but without ever executing a step that might bring two processes to divergent decisions, thereby preserving safety.

The liveness part is a little more complicated because it must deal directly with the main feature that characterizes our model: the presence of dynamic and transient message omissions. It must be guaranteed that the processes receive enough messages to make progress despite omissions. The transient aspect ensures that this requirement is met: processes may be unable to communicate properly for an arbitrarily long period of time, during which the best they can do is to avoid taking wrong decisions (perhaps stubbornly probing the channel's status), but eventually the network conditions get better to allow them to complete the task. The dynamic aspect is instead aimed at generalizing the particular pattern that the omissions can have (e.g., distributed among the processes or concentrated to few ones). The bound on omissions to ensure progress, no matter their pattern, is $f \leq \lceil \frac{n}{2} \rceil (n-k) + k - 2$ (if all the processes must decide, then $k = n$ and $f \leq n-2$). Since our algorithm is an extension, such bound holds also in our case, but we refer to [5] for a more complete and formal explanation of liveness.

A. Safety

Theorem 1 (Validity)**.** *If all processes propose the same value v, then any process that decides, decides v.*

Proof Sketch: Each process has its proposal initialized to the same value v. After the (perhaps repeated) message exchange, as all the messages contain the same value v, processes get the same (weak) majority in the pre-prepare phase, thus setting their local proposal to the same majority value (value set in the box). Then, they wait to receive more than $\frac{n}{2}$ messages for the prepare phase (line 14), where they get the same strong majority thus setting the same value (line 18). Subsequently, they again wait for more than $\frac{n}{2}$ messages, all of them with the same value v, and they decide on v (lines 24, 27 and 35). ■

Theorem 2 (Agreement)**.** *No two processes decide differently.*

Proof Sketch: By contradiction, suppose they take different decisions, respectively on v_0 and v_1. This cannot happen when a process catches up with another one that is executing a later phase, by copying its state (line 9). So, there are two cases.

1) They decide in the same phase. Then it must be that they received a strong majority of messages (line 23) for the decided values with the same phase number ϕ. So, it must be that at least one process broadcast two messages with two different proposals. As all processes follow the algorithm faithfully, this is clearly a contradiction.

2) They decide in different phases. In this case, one should notice that in the prepare phase each process sets its proposal either to a value contained in a strong majority of the received messages (in that phase), or to a default value $\perp$. So, in the decision phase, the only values allowed are the default $\perp$ and a single binary value, either 0 or 1 (but not both of them). The first process that decides on a value, receives a strong majority for that value in the decision phase. Therefore, all the other processes in the same phase receive at least one message with that value. As a consequence, any other process can either decide that value in that phase, or can just set its proposal to that value (line 27, without tossing a coin). The decision of a process thus *locks* a particular value, which is the only one that can be proposed and decided by all processes in subsequent phases. For this reason, the process supposed to take a different decision is a clear contradiction.

■

B. Liveness

In the next, we say that a process can *make progress*, each time it is able to execute either line 10 or line 32, a new phase. In the first case, the process makes progress when it receives at least one message carrying a phase number higher than its, by copying all the information to its state. In the second, it receives a set of messages from more than half of the processes carrying the same phase number of its.

Lemma 1. *If a process has phase ϕ, then there are more than $\frac{n}{2}$ processes with phase at least $\phi - 1$.*

Proof Sketch: Consider the first process that sets its phase to value ϕ. Then it must have received more than $\frac{n}{2}$ messages with phase $\phi - 1$. Hence there are more than $\frac{n}{2}$ processes with phase at least $\phi - 1$ ■

Lemma 2. *If a process has phase ϕ, then there will be at least $k > \frac{n}{2}$ processes with phase at least $\phi - 1$.*

Proof Sketch: By lemma 1 more than $\frac{n}{2}$ processes (call their set A) have phase at least $\phi - 1$, we have to show that eventually also $k - |A|$ other processes reach that phase. Suppose by contradiction that they cannot. Then all messages from A to all the other processes must be dropped. In this setting, we can rewrite the number of processes as composed by the following groups: $n = |A| + (k - |A|) + (n - k)$. Therefore the number of omissions should be

$$\begin{aligned}
|A|(n-k)+|A|(k-|A|) &> \frac{n}{2}(n-k)+|A|(k-|A|) && (1)\\
\text{for } k-|A|,|A|>0 \qquad &\geq \frac{n}{2}(n-k)+(|A|+k-|A|-1) && (2)\\
&= \frac{n}{2}(n-k)+k-1 && (3)\\
&> f && (4)
\end{aligned}$$

As the number of omissions exceeds the bound given above, this is a contradiction, so eventually k processes have phase at least $\phi - 1$. ■

Lemma 3. *If at least $k > \frac{n}{2}$ processes have phase at least ϕ, then some process in the system must eventually increase its phase value.*

Proof Sketch: Our system can be described by three sets. A is the set of processes with phase greater than ϕ, B is the set of those with phase ϕ, and C is the set of those with phase smaller than ϕ. The cardinality of each set is a, b, c respectively, such that: $a + b = k$. By contradiction, suppose that no process ever increases its phase. Therefore: 1) no message from A must reach either B or C; 2) no message from B must reach C; 3) if there are more than $\frac{n}{2}$ processes in either A or B, they must be prevented from exchanging enough messages among themselves. For what concerns 1) and 2), the number of omissions that must be imposed is at least $ab + ac + bc = c(a + b) + ab = (n - k)k + ab > \frac{n}{2}(n - k) + ab$. Now, let us compute a bound for ab. If either A or B is empty, then $ab = 0$, but the other set contains $k > \frac{n}{2}$ processes that, by exchanging messages among themselves can still make progress. So, in this case, at least k omissions (one for each process when k is minimal) are necessary. If otherwise none of them is empty, then $a, b > 0$ so $ab > a+b-2 = k-2$. In either case, the total number of omissions required exceeds the bound f given above. This is a contradiction, so eventually some process increases its phase value.

■

Lemma 4 (Progress)**.** *If ϕ is the highest phase reached in the system, then eventually some process sets its phase to $\phi + 1$.*

Proof Sketch: As there is at least one process with phase ϕ, according to lemma 2, there will eventually be $k > \frac{n}{2}$ processes with phase at least $\phi - 1$. Our system can thus be described by three sets. A is the set of processes with phase ϕ, B is the set of those with phase $\phi - 1$, and C is the rest of the system. The cardinality of each set is a, b, c respectively, such that: $a > 0$, $a+b = k$. According to lemma 3, since there are k processes with phase at least $\phi - 1$, some process must increase its phase value. In the worst-case, by applying the lemma repeatedly, we have that C gets empty, so processes in B must necessarily start increasing their phase value. When B gets too small, $|B| < k$, we can apply lemma 3 to processes in A, that all have phase ϕ. Again, in the worst-case, we have to wait for B to become empty. Anyway, as soon as all processes have phase ϕ, at least one of them must eventually set its phase value to $\phi+1$.

■

This lemma is very important because it proves that, as soon as the bound on the omissions is respected, the processes are guaranteed to be able to make progress, thereby reaching an arbitrarily high phase. However, it is important to recall from the impossibility results in [2], [3] that making progress is not a sufficient condition for the processes to reach consensus. Indeed, they demonstrate that processes can make progress infinitely often, by touring across bivalent system states, in which no safe decision can ever be taken. What makes consensus possible (i.e. terminate) is randomization, and the termination property is sketched in the following theorem.

Theorem 3 (Termination)**.** *At least k processes eventually decide with probability 1.*

Proof Sketch: By contradiction, suppose that no process ever takes a decision, but everyone can make progress (see lemma 4) infinitely often. In the decision phase, each process can either set its proposal to a binary value present in some message or to a random one, by tossing a coin. By means of arguments provided in previous proofs in relation to the quorum of messages received, even if no decision takes place, it is only one (if any) the binary value that each process is allowed to set in the decision phase. So, we can partition the processes into two groups: (A) processes that set the same value, (B) processes that toss a coin. If B is empty, then the processes set the same value v. Since v becomes the only value proposed, it is the only one that can be decided in the next phases. By lemma 2, 3 and 4, there are at least k processes able to make progress, so at least k that reach a decision. If B is not empty, then there is a non-zero probability that these processes get the same binary value of the processes in A, after tossing a coin, with $p = 2^{-|B|}$. Since processes make progress infinitely often, if (by assumption) they do not decide, they eventually have to toss a coin infinitely often. However, the probability of not setting the same value is asymptotically $\lim_{phase \to \infty}(1-p)^{phase} = 0$. Hence eventually at least k processes decide. ■

VII. 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 [21]. 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 increase the 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 IV), 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

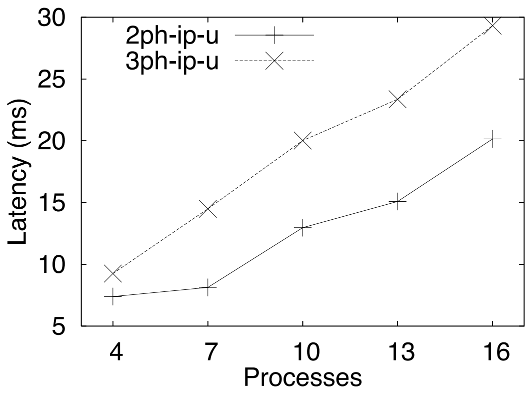


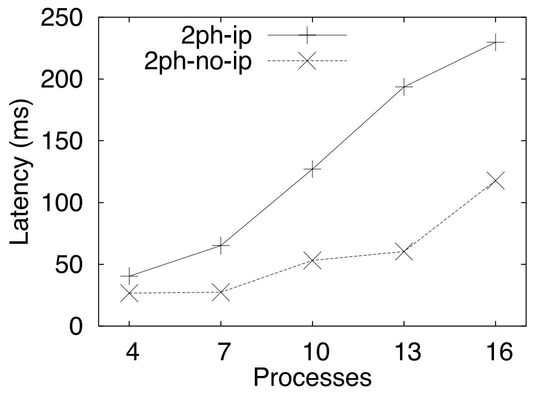
Figure 1: Latency with uniform proposal values.

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 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).

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 (line 29, with or without *immediate progress*) after running the first two or three phases, depending on the protocol version. Now, let $V[i]$, $0 \leq i \leq n-1$ be a vector with the new proposals and consider the event



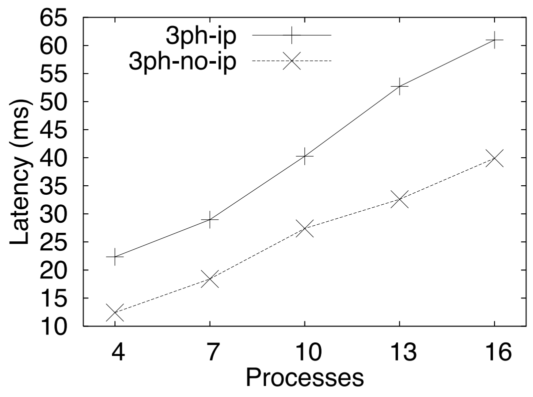
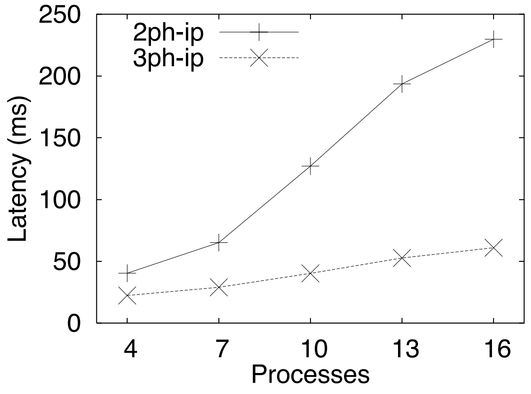


Figure 2: Evaluation of the *smart-receive()* strategies.

$\mathcal{E} := \{\sum_{i=0}^{n-1} V[i] \ equals \ \sum_{i=0}^{n-1} 1 - V[i]\}$. 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.

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)



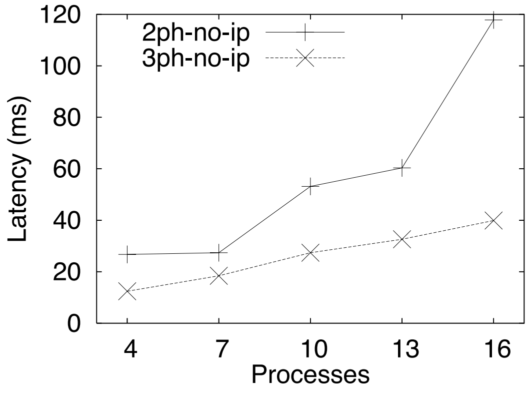


Figure 3: The impact of the third phase.

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. Although 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 in few rounds.

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: $\mathcal{P}_s = 0.1$ and $\mathcal{P}_r = 0.3$, abbreviated as adversary *adv-1-3*; and $\mathcal{P}'_s = 0.3$ and $\mathcal{P}'_r = 0.6$, abbreviated as adversary *adv-3-6*. According to these probability, defining the event $\mathcal{E} := \{\text{message reaches destination}\}$, $\mathcal{P}[\mathcal{E}] = (1 - \mathcal{P}_s)(1 - \mathcal{P}_r) = 0.63$ and $\mathcal{P}'[\mathcal{E}] = (1 - \mathcal{P}'_s)(1 - \mathcal{P}'_r) = 0.28$. 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*.

A curious latency trend is visible in the figures, 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

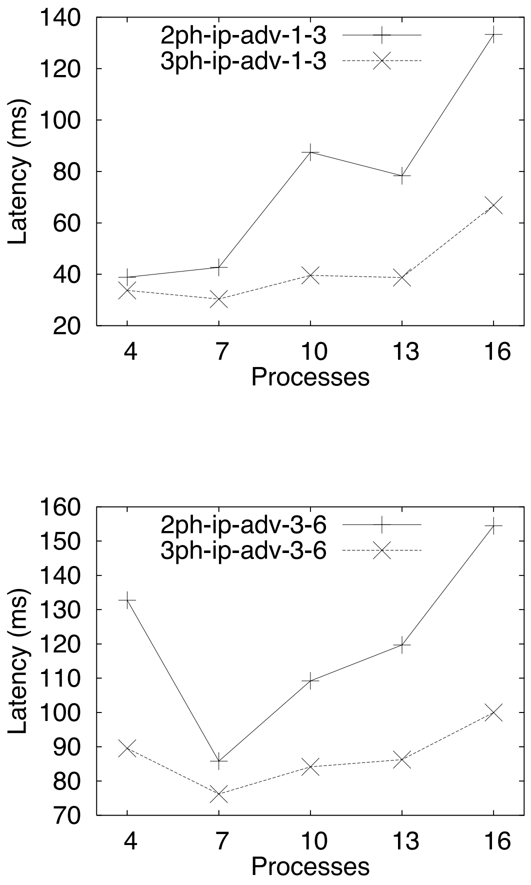


Figure 4: *Immediate Progress* in presence of an adversary.

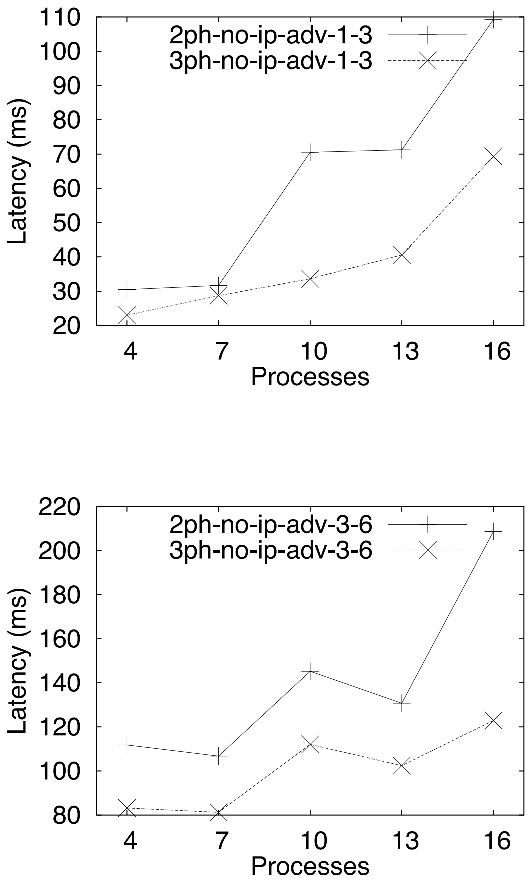


Figure 5: *No Immediate Progress* in presence of an adversary.

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 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.

C. *An interesting relationship*

In [22], a deterministic version of a randomized wait-free shared-memory consensus algorithm is proposed and analyzed. The intention of the paper is to show that randomization helps even if it is not present inside the algorithm but rather in the environment itself. More precisely, the system is not asynchronous but each process takes a random time (characterized by a random variable X, different for each process) to complete each operation. The result holds also if these random variables do not have a finite expectation.

As for every binary consensus algorithm, each process prefers either 0 or 1. The decision is settled by means of a race. There are two arrays (each one representing a binary proposal 0 or 1) of atomic read/write registers, accessible to all processes, starting from their first position. The computation of each process proceeds in rounds, whose number is used as offset to address the correct register position in the array. At every round, a process computes the location of the registers in the two arrays. If one of these registers has a bit raised, the process updates its proposal to the value that the array represents. Otherwise it raises a bit

TABLE I: AVERAGE NUMBER OF ROUNDS IN WHICH 16 PROCESSES REACH CONSENSUS IN A 802.11B NETWORK AND CONFIDENCE LEVEL OF 95%.

Group Size = 16	Average Termination Round ± Confidence Interval
2ph-ip	17.80 ± 1.48
2ph-no-ip	8.99 ± 0.49
2ph-ip-adv-1-3	10.23 ± 0.63
2ph-ip-adv-3-6	7.21 ± 0.35
2ph-no-ip-adv-1-3	6.60 ± 0.23
2ph-no-ip-adv-3-6	6.00 ± 0.15
3ph-ip	6.85 ± 0.28
3ph-no-ip	4.60 ± 0.16
3ph-ip-adv-1-3	5.50 ± 0.23
3ph-ip-adv-3-6	4.90 ± 0.21
3ph-no-ip-adv-1-3	4.60 ± 0.18
3ph-no-ip-adv-3-6	4.30 ± 0.13

in the register of the array corresponding to its proposal. At this point, of the two registers localized at the same position p (indicated by the round/offset) in each array, only one has a bit raised, and the process has its proposal set to the array to which this register belongs. Then the process checks if the register in position $p-1$ of the array representing the other proposal is not set. If it is not, it can safely infer that its computation has fallen ahead the one of the other processes, and can safely decide on its own proposal. Otherwise it increases the round number and repeats the procedure.

This algorithm obviously does not work if processes run in lock-step (synchronized) execution, because none of them may eventually be able to impose its proposal to the others. However, by leveraging on the random execution time of the operations, it is likely that the computation of the processes gets dispersed in time and space. In particular, as proven in [22], there is one process (the leader) that emerges faster than the others, decides on its proposal, and makes the others (learners) set and decide on the same value.

Our model and algorithm are clearly very different from the ones just presented. Yet we can provide evidence of a very simple relationship. The crucial part in our algorithm is due to the lines 9-13, where a process, once it has received a message carrying a later phase, just copies (learns) the sender's state and possibly also its decision. Therefore, if a particular process is allowed to get/send messages and to make progress immediately, it would (unconsciously) rise to a leader position, dictating its proposal to the other processes. In our practical situation, the more processes start a phase with the same proposal, the more likely is for them to get good quorums and decide immediately.

This relationship is evidenced in the evaluation section, particularly in presence of an adversary. By dropping messages, the adversary indeed worsen the network situation, but makes more probable that few lucky processes come up being faster. By looking at the latency results, it can be seen that the 2-phase algorithm turns out to be quicker under bad network conditions. This may not be noticed for the 3-phase version, because of its added time complexity and the need to rebroadcast messages. Nevertheless, it is observable by dealing with a different notion of speed, namely the number of rounds needed to reach consensus. In this case, it is clear (see Table I) that this relationship indeed exists, and that message dropping can be leveraged either to make a leader rise quickly, or at least to reduce the number of processes involved in the decision process, and so its complexity.

VIII. 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 processes is being used and if algorithms are carefully engineered to be as efficient as possible. A protocol previously presented 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 about the state of the system 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. Also, we discuss an interesting connection with a different protocol, which was formally proved to be fast. 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 EC through project FP7-257475 (MASSIF), and by the FCT through the Multiannual and the CMU-Portugal Programmes, and the project PTDC/EIA-EIA/113729/2009 (SITAN).

REFERENCES

[1] Bruno Vavala, Nuno Neves, Henrique Moniz, and Paulo Veríssimo. Randomized consensus in wireless environments: a case where more is better. In *Proceedings of the 3rd International Conference on Dependability (DEPEND)*, pages 7–12, 2010.

[2] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[3] Nicola Santoro and Peter Widmeyer. Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 304–313, 1989.

[4] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[5] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization can be a healer: consensus with dynamic omission failures. In *Proceedings of the 23rd International Conference on Distributed Computing (DISC)*, pages 63–77, 2009.

[6] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

[7] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

[8] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

[9] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[10] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo. Experimental comparison of local and shared coin randomized consensus protocols. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 235–244, 2006.

[11] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, Antonio Casimiro, and Paulo Verissimo. Intrusion tolerance in wireless environments: An experimental evaluation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 357–364, 2007.

[12] P. Urban, X. Defago, and A. Schiper. Chasing the flp impossibility result in a lan or how robust can a fault tolerant server be? In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 190 –193, 2001.

[13] Michel Raynal and Matthieu Roy. A note on a simple equivalence between round-based synchronous and asynchronous models. In *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 387–392, 2005.

[14] Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3):232–249, 2007.

[15] U. Schmid, B. Weiss, and I. Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

[16] Y. Moses and S. Rajsbaum. A Layered Analysis of Consensus. *SIAM Journal on Computing*, 31(4):989–1021, April 2002.

[17] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[18] H. Moniz, N.F. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 537–546, 2010.

[19] Monica Dixit and Antonio Casimiro. Adaptare-FD: A Dependability-Oriented Adaptive Failure Detector. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 141–147, 2010.

[20] Rachid Guerraoui and Andre Schiper. Consensus: The big misunderstanding. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, pages 183 –188, 1997.

[21] 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 *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, 2002.

[22] James Aspnes. Fast deterministic consensus in a noisy environment. *Journal of Algorithms*, 45(1):16–39, 2002.