Turquois: Byzantine Consensus in Wireless Ad hoc Networks

Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia
University of Lisboa*
Portugal
{hmoniz, nuno, mpc}@di.fc.ul.pt

Abstract

The operation of wireless ad hoc networks is intrinsically tied to the ability of nodes to coordinate their actions in a dependable and efficient manner. The failure of some nodes and momentary breakdown of communications, either of accidental or malicious nature, should not result in the failure of the entire system. This paper presents Turquois - an intrusion-tolerant consensus protocol specifically designed for resource-constrained wireless ad hoc networks. Turquois allows an efficient utilization of the broadcasting medium, avoids synchrony assumptions, and refrains from public-key cryptography during its normal operation. The protocol is safe despite the arbitrary failure of $f < \frac{n}{3}$ processes from a total of n processes, and unrestricted message omissions. The protocol was prototyped and subject to a comparative performance evaluation against two well-known intrusion-tolerant consensus protocols. The results show that, as the system scales, Turquois outperforms the other protocols by more than an order of magnitude.

1. Introduction

Intrusion tolerance lets systems remain operational, even if some of their components fall under the control of a sophisticated adversary and start to act with malicious intent. Over the years, several solutions based on this concept have been proposed for LAN settings, for example, to build replicated services and group communication protocol stacks (e.g., [3, 22]). However, very little work has been done to develop distributed system models and protocols for intrusion-tolerant wireless ad hoc networks.

Wireless ad hoc networks are characterized by the lack of centralized control. There is no notion of infrastructure and usually every node plays an equal role on the network operation. These characteristics make them particularly suited for unplanned or emergency scenarios, where the reliance on a single point of failure is not only inappropriate but maybe even unattainable.

Nevertheless, the ability for nodes to conduct coordinated activities is of paramount importance in many wireless ad hoc network applications. Nodes may need, for instance, to synchronize clocks, order messages, elect a leader, accommodate other nodes in a group, or agree on a common decision. All these activities require some sort of agreement among the nodes, and therefore it is imperative that this operation is performed in a dependable way. The failure of some nodes should not be synonymous with the failure of the entire system. Hence, correct nodes should be able to reach agreement even if others are uncooperative, either by crashing, not communicating, or plainly acting selfishly or maliciously.

The consensus problem is a fundamental abstraction of this necessity for agreement. Basically, it states that every node proposes a value, and then the nodes have to decide on a common result. While simple to describe, it is far from being a trivial problem. Consensus has associated impossibility results in systems where nodes or communication links can fail [13, 25]. Wireless ad hoc networks, in particular, are inherently unreliable. Environmental phenomena such as interference, fading, and collisions give rise to pervasive communication failures, and node mobility may result in momentary disconnection. In addition, wireless ad hoc networks are usually resource-constrained. They usually have less bandwidth than wired local networks, and the computational power of their nodes is often more restricted.

The aim of this paper is to conciliate intrusion tolerance with the unreliable resource-constrained nature of ad hoc networks. The paper focuses on the problem of binary consensus for single-hop wireless ad hoc networks, assuming that nodes are subject to transitory disconnection and permanent corruption by a malicious entity. In order to maximize the efficiency of the solution, we will make a rational use of the resources provided by the environment. Namely, since the network provides a natural broadcasting medium, the cost of transmitting a message to multiple nodes can be

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

just the same of sending it to a single one, as long as they are within communication range. Properly exploited, this property can have a profound impact on performance. However, to take advantage of this property one needs to depart from the traditional modeling assumptions of intrusion-tolerant systems. Usually they assume a reliable point-to-point communication model, which hinders any possibility of taking advantage of the broadcasting medium (because it forces the implementation of end-to-end message delivery mechanisms, e.g., TCP). Thus, the underlying model should also embrace the inherent unreliability of radio communications.

In this paper, we propose a model that derives from the *communication failure model* introduced by Santoro and Widmayer [25]. Their model assumes the existence of dynamic and transient transmission faults, meaning that any communication from one node to another can be faulty at one moment and be correct at another. In a wireless environment, this implies that any broadcast message may be delivered non-uniformly by the intended recipients. Some of them may deliver the message, while others might not. Under particularly harsh conditions, like a jamming attack, even all messages may be lost during a period of time.

Our model assumes a system composed of n ad hoc nodes where a subset f of them may be compromised by a malicious adversary (with $f < \frac{n}{3}$). Compromised nodes can fail in an arbitrary (or Byzantine) manner, namely by sending messages with erroneous content or by simply becoming silent. Therefore, we will consider that potentially all transmissions from these nodes might be lost (or discarded), either due to network omission faults or bad behavior. Additionally, we will assume the existence of dynamic omission transmission faults that might affect the communications between correct nodes. Our consensus protocol will ensure progress towards a decision in rounds where these faults are $\sigma \leq \lceil \frac{n-t}{2} \rceil (n-k-t) + k - 2$ (where k is the number of nodes required to decide and $t \leq f$ is the number of processes that are actually faulty). If a higher number of faults occur, then the protocol always ensures safety, but progress might be stopped until the network starts to behave better.

The paper has the following contributions:

(1) A binary consensus protocol, named Turquois[1], designed to tolerate a combination of Byzantine nodes and dynamic omission transmission faults. To the best of our knowledge, this is the first consensus protocol that exhibits this characteristic.

(2) Since the system is asynchronous and can have both Byzantine nodes and dynamic omission faults, consensus is bound by the impossibility results of [13, 25]. Turquois circumvents these impossibility results by employing randomization, ensuring termination with probability 1.

[1]Turquois: 1. a semiprecious stone, typically opaque and of a sky-blue color; 2. french for Turk, historic enemy of the Byzantine.

(3) A novel mechanism for broadcast message authentication that resorts to an inexpensive hashing operation instead of typical public-key cryptography, preserving the computational restrictions usually associated with mobile nodes and increasing efficiency.

(4) An experimental evaluation of the protocol and a comparison with two well-known intrusion-tolerant binary consensus protocols [7, 8] in various network and failure scenarios. The evaluation shows that our protocol performs significantly better (more than one order of magnitude faster in several cases), exposing the inappropriateness of the classical model of intrusion-tolerant systems for wireless ad hoc networks.

2. Related Work

Over the past decade, there have been some contributions to the solution of consensus in wireless ad hoc networks, however, almost all of them did not consider the presence of Byzantine nodes. Research on Byzantine fault-tolerant protocols for wireless environments has been practically restricted to broadcasting problems [14, 20, 12, 4, 15].

Concerning the problem of consensus, Badache et al. were the first to present a protocol specifically for wireless environments [1]. Their solution considers that mobile hosts (MHs) are connected to fixed mobile support stations (MSSs), which are assumed to be fully connected. To solve consensus, each MH communicates the initial proposal value to the respective MSS. The MSSs execute amongst themselves the Chandra-Toueg consensus protocol using a $\diamond S$ failure detector [9], and then communicate the decision value to the associated MHs. Later on, this work was extended by Seba et al. to take into consideration the dynamism in the set of MSSs executing consensus due to the handover of MHs [27].

Wu et al. describe a hierarchical consensus protocol for mobile ad hoc networks [30]. Their protocol selects a subset of predefined mobile nodes to act as *clusterheads*, which take essentially the same role of the MSSs in the protocol of Badache et al. [1]. The *clusterheads* gather the initial values of their associated nodes and execute consensus using a $\diamond P$ failure detector. The decision is then propagated from the *clusterheads* to the nodes. Vollset et al. present randomized consensus protocols that tolerate crashing nodes and arbitrary topological changes [28]. Their solution, however, requires a fairness condition where correct nodes are not permanently disconnected.

The research discussed so far assumes reliable links. In this paper, we address the consensus problem under a model that extends the *communication failure model* of Santoro and Widmayer [25, 24]. The communication failure model has an associated impossibility result, stating that there is no fixed-time solution to the problem of *k-agreement* (i.e.,

$k > \lceil \frac{n}{2} \rceil$ nodes decide the same value 0 or 1) if more than $n-2$ links are allowed to lose messages at every synchronous communication step. This result ends up being very restrictive because a single node crash causes n omission failures per round, thus preventing consensus.

Chockler et al. described a consensus algorithm for a system where nodes can fail by crashing and messages can be lost due to collisions [11]. Their protocol can solve consensus due to the additional power offered by collision detectors, which allow nodes to take measures to recover from message losses. Message omissions other than those due to collisions, however, are not covered by their model.

Borran et al. [6] address consensus under the *heard-of model* (HO) [10], which permits a fine-grained specification of the patterns of message delivery allowed for the problem to be solvable. This work uses the HO model to express the Paxos algorithm [16] and extend it with a communication layer for wireless networks, which provides a leader election service. The reliance on a leader may not be very appropriate in some ad hoc scenarios, and the problem of dynamic omission transmission failures is not taken in consideration because the protocol assumes periods of reliable and delay bounded message deliveries.

Biely et al. also employs the HO model to distinguish cases where the fault pattern exceeds the upper bound of Santoro and Widmayer, but not in a harmful way to the system (e.g., $n-1$ faults are harmful if they originate at the same node, but may not be if they originate each one at a different nodes) [5]. The work of Schmid et al. presents an analogous contribution in the sense that it limits the number of faults that each node may experience [26]. None of these two contributions, however, deal with the essence of the Santoro-Widmayer impossibility result because faults are artificially restricted.

Moniz et al. address this issue by presenting a randomized consensus algorithm that tolerates a new upper bound of $\lceil \frac{n}{2} \rceil (n-k) + k - 2$ omission link failures per communication round, regardless of their pattern [19]. The protocol described in this paper tolerates not only dynamic transmission omission faults, but also a static, *a priori* unknown, subset of Byzantine nodes.

3. System Model

The system is composed by a fixed and known set of n nodes, each one running a single process belonging to $\Pi = \{p_0, p_1, ..., p_{n-1}\}$. The communication between processes proceeds in asynchronous broadcast rounds. At each round, every process $p_i \in \Pi$ transmits a message m to every process $p_j \in \Pi$, including itself, by invoking `broadcast(`m`)`. A round r is defined as the r^{th} time that processes invoke the `broadcast()` primitive and is triggered by a clock tick local to each process.

The fault model assumes that up to f processes can be Byzantine, and that these processes may fail in an arbitrary way. For example, a Byzantine process can become silent, send messages with wrong values, or collude with other Byzantine processes to disrupt the correct operation of the system. Such processes are said to be *faulty*, while processes that follow the algorithm are called *correct*.

The fault model also accommodates dynamic omission failures in message transmissions amongst correct processes. A transmission between two correct processes p_i and p_j is subject to an omission failure if the message broadcast by p_i is not received by p_j. The number of omission failures that can occur per round is unrestricted, in the sense that safety properties are always guaranteed. However, in order to ensure progress, we will make the following fairness assumption: given an unbounded number of rounds, there are infinitely many rounds in which the number of omission faults that affect correct processes is bounded by a σ value (see protocol description). If a message m transmitted by process p_i to process p_j is not subject to a dynamic omission failure and both processes are correct, then m is eventually received by p_j.

Cryptographic functions employed in the protocol are secure and can not be subverted by an adversary, and each process $p_i \in \Pi$ can call a local random bit generator to obtain unbiased bits observable only by p_i.

4. Problem Definition

The paper addresses the k-*consensus* problem. This 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 of them have to decide on a common value proposed by one of the processes (with $\frac{n+f}{2} < k \leq n-f$). The remaining non-Byzantine processes (at most $n-k$) 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 correct processes propose the same value v, then any correct process that decides, decides v.

Agreement. No two correct processes decide differently.

Termination. At least k correct processes eventually decide with probability 1.

5. Turquois: Byzantine k-Consensus

The algorithm Turquois allows k processes out of n to reach consensus on a binary value $v \in \{0, 1\}$ (see Algorithm 1). Correctness in maintained as long as the number of Byzantine processes is bounded by $f < \frac{n}{3}$. Furthermore,

the algorithm ensures safety (i.e., the validity and agreement properties) despite an unrestricted number of transmission omission faults. Progress towards termination is guaranteed in rounds where the number of omission faults is $\sigma \leq \lceil \frac{n-t}{2} \rceil (n-k-t)+k-2$, where $t \leq f$ is the number of processes in the system that are actually faulty. Turquois is a randomized algorithm. It relies on each process p_i having access to a *local coin*[2] mechanism that returns random bits observable only by p_i (e.g., [2, 7]). The first local coin protocol was proposed by Ben-Or, of which our protocol is reminiscent [2].

In the algorithm, each processes p_i has an internal state comprised by three variables: (1) the *phase* $\phi_i \geq 1$, (2) the *proposal value* $v_i \in \{0, 1\}$, and (3) the *decision status* $status_i \in \{decided, undecided\}$. Each process starts its execution with $\phi_i = 1$, $status_i = undecided$, while v_i is set to the initial proposal value indicated by the input parameter $proposal_i$ (Lines 1-3).

The algorithm runs in cycles of three phases, which are called CONVERGE, LOCK, and DECIDE. A process is in each one of these phases when its phase value is, respectively, $\phi_i \pmod 3 = 1$, $\phi_i \pmod 3 = 2$, and $\phi_i \pmod 3 = 0$. In the CONVERGE phase, processes try to converge their proposal values by assuming the value they observe the most. In the LOCK phase, processes try to *lock* on a single value $v \in \{0, 1\}$. Each process either sets its proposal value to this v or to a value $\perp$ indicating a lack of preference. Finally, in the DECIDE phase, processes attempt to decide on the value *locked* on the previous phase. If a process is not able to decide at the end of a DECIDE phase, it may propose a random value at the beginning of the following cycle. This random step guarantees that eventually there is a cycle that starts with every correct process proposing the same value. When this happens, k correct processes necessarily decide by the end of that cycle.

The algorithm is run in parallel by tasks T1 and T2, which are activated by the respective **when** condition (Lines 5 and 8). When activated, a task runs towards completion without any interruption from the other task. Task T1 defines a broadcasting round and is activated periodically upon a local clock tick (Lines 5-7). A process p_i broadcasts a message of the form $\langle i, \phi_i, v_i, status_i \rangle$ containing its identifier i and the variables that comprise its internal state.

Task T2 is activated whenever a message arrives (Lines 8-43). Some of the messages that a process is supposed to receive may be lost, or may carry invalid content if transmitted by a Byzantine process. Therefore, all arriving messages are subject to a validation procedure that constrains the wrongful actions of Byzantine processes. Essentially, a message is considered *valid* if it could have been sent by a process that followed the algorithm (details in Section 6). Valid messages are accumulated in a set V_i (Line 9), while the others are discarded.

Based on its current internal state and the messages accumulated in set V_i, a process p_i performs a state transition, which happens when one of two conditions occur:

1. the set V_i holds some message whose phase value ϕ is *higher* than the current phase ϕ_i of p_i;

[2]As opposed to a *shared coin* that returns bits observable by all processes (e.g., [21, 8]).

Algorithm 1: Turquois: a Byzantine k-consensus algorithm

Input: Initial binary proposal value $proposal_i \in \{0, 1\}$
Output: Binary decision value $decision_i \in \{0, 1\}$

1 $\phi_i \leftarrow 1$;
2 $v_i \leftarrow proposal_i$;
3 $status_i \leftarrow undecided$;
4 $V_i \leftarrow \emptyset$;

TASK T1:
5 **when** *local clock tick* **do**
6 | broadcast($\langle i, \phi_i, v_i, status_i \rangle$);
7 **end**

TASK T2:
8 **when** $m = \langle j, \phi_j, v_j, status_j \rangle$ *is received* **do**
9 | $V_i \leftarrow V_i \cup \{m : m$ is *valid*$\}$;
10 | **if** $\exists \langle *, \phi, v, status \rangle \in V_i : \phi > \phi_i$ **then**
11 | | $\phi_i \leftarrow \phi$;
12 | | **if** $\phi \pmod 3 = 1$ ***and*** v *is the result of a coin flip* **then**
13 | | | $v_i \leftarrow \texttt{coin}_i()$;
14 | | **else**
15 | | | $v_i \leftarrow v$;
16 | | **end**
17 | | $status_i \leftarrow status$;
18 | **end**
19 | **if** $|\{\langle *, \phi, *, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n+f}{2}$ **then**
20 | | **if** $\phi_i \pmod 3 = 1$ **then** /* **phase CONVERGE** */
21 | | | $v_i \leftarrow$ majority value v in messages with phase $\phi = \phi_i$;
22 | | **else if** $\phi_i \pmod 3 = 2$ **then** /* **phase LOCK** */
23 | | | **if** $\exists v \in \{0, 1\}$: $|\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n+f}{2}$ **then**
24 | | | | $v_i \leftarrow v$;
25 | | | **else**
26 | | | | $v_i \leftarrow \perp$;
27 | | | **end**
28 | | **else** /* **phase DECIDE** */
29 | | | **if** $\exists v \in \{0, 1\}$: $|\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n+f}{2}$ **then**
30 | | | | $status_i \leftarrow decided$;
31 | | | **end**
32 | | | **if** $\exists v \in \{0, 1\}$: $|\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| \geq 1$ **then**
33 | | | | $v_i \leftarrow v$;
34 | | | **else**
35 | | | | $v_i \leftarrow \texttt{coin}_i()$;
36 | | | **end**
37 | | **end**
38 | | $\phi_i \leftarrow \phi_i + 1$;
39 | **end**
40 | **if** $status_i = decided$ **then**
41 | | $decision_i \leftarrow v_i$;
42 | **end**
43 **end**

2. the set V_i holds more than $\frac{n+f}{2}$ messages whose phase is *equal* to the current phase ϕ_i of p_i.

The first case is simpler (Lines 10-18). When the condition is met (Line 10), process p_i updates the state to match the state of the received message, with a slight exception. The special instance is the following: if the phase value is $\phi \pmod 3 = 1$ and the value v was obtained from the result of a coin flip (which can be verified from the validation procedure described in Section 6), then p_i executes a local coin flip to determine v_i (Lines 12-13). Since it is not possible to force Byzantine processes into a fair coin flip, this step becomes necessary to guarantee that correct processes assume a random value.

The second case is more complex (Lines 19-39). The way a process p_i updates its state depends on the value of its current phase number ϕ_i modulo 3. In CONVERGE phases ($\phi_i \pmod 3 = 1$) the proposal value is set to the majority value of all messages with phase value $\phi = \phi_i$ (Lines 20-21).

In LOCK phases ($\phi_i \pmod 2 = 2$) the proposal value v_i is updated the following way (Lines 22-27): if there are more than $\frac{n+f}{2}$ messages of the form $\langle *, \phi, v, * \rangle$ in V_i with $\phi = \phi_i$ and the same value v, then v_i is set to v (Lines 23-24), otherwise it is set to a special value $\perp \notin \{0, 1\}$ indicating a lack of preference (Lines 25-26). This step ensures that in the following phase $\phi_i + 1$ every process either proposes the same value $v \in \{0, 1\}$ or $\perp$. Furthermore, if there was unanimity amongst correct processes at the previous phase $\phi_i - 1$, then every process must set its proposal value to the same value v (since messages with a different value are considered invalid). This will imply that in the next phase $\phi_i + 1$ every process receives the same value $v \in \{0, 1\}$ in all valid messages and decides.

In DECIDE phases ($\phi_i \pmod 2 = 0$), a process sets $status_i$ to $decided$ if there are more than $\frac{n+f}{2}$ messages of the form $\langle *, \phi, v, * \rangle$ in V_i with $\phi = \phi_i$ and the same value $v \neq \perp$ (Lines 29-31). The proposal value v_i is set to v if there is at least one message of the form $\langle *, \phi, v, * \rangle$ in V_i with $\phi = \phi_i$ and a value $v \neq \perp$. Otherwise, v_i is set to the value of function `coin()`, which returns a random number 0 or 1, each with probability $\frac{1}{2}$ (Lines 32-36). Regardless of the previous steps, the phase is always incremented by one unit (Line 38).

At the end of each round, a process p_i checks if $status_i$ has been set to $decided$. If so, it decides by setting the output variable $decision_i$ to the current proposal value v_i (Lines 40-42). Further accesses to this variable do not modify its value. Hence, they have no impact on the correctness of the algorithm. The full correctness proof can be found in an accompanying technical report [18].

6. Validation of Messages

A process p_j must check the validity of arriving messages before adding them to set V_j. This procedure is fundamental to the correct operation of the protocol because it limits the wrongful actions that a Byzantine process can accomplish. There are two types of validation that a message must pass: authenticity validation and semantic validation. The first guarantees that some of the fields of a message were actually generated by a process p_i, while the second ensures that the contents of a message are congruent with the current execution of the algorithm. A message is deemed *valid* if it passes both tests.

6.1. Authenticity Validation

This form of validation provides (partial) message authentication. More precisely, for any message $\langle i, \phi, v, status \rangle$, it provides to a receiving process p_j assurance that the values of ϕ and v originated at the alleged source process p_i. This statement deserves the following caveat. The authenticity of the $status$ variable is not protected by this mechanism. Consequently, it is possible for a malicious entity to replay a message $\langle i, \phi, v, status \rangle$ with an arbitrary $status$ value. This, however, does not impact the correctness of the protocol because our semantic validation mechanism (see next section) requires processes to justify their $status$ based on the received proposal values, therefore, making the attack ineffective.

Authentication is based on a mechanism for generating and verifying *one-time hash-based message signatures* that is particularly efficient for a round-based group communication protocol with a small domain of input values. In our case, the mechanism is devised for an input domain of three values (0, 1, and $\perp$), which represents the possible proposal values that a message can have. To the best of our knowledge, this is the first time such a mechanism is employment in an agreement protocol.

The mechanism is composed by a generic *message authentication* procedure for each phase of the k-consensus protocol, and by a *key exchange* procedure that has to be executed periodically. The message authentication resorts to an efficient one-way hash function H to generate hash values of length h (e.g., SHA-256 or RIPEMD-160) [17]. The key exchange procedure resorts to a more computationally expensive trapdoor one-way function F (e.g., RSA [23]) that is used to sign an array of verification keys. It is assumed that each process p_i has an associated public/private key pair to be used in F, where pu_i is the public key and pr_i is the private key. Every process knows the public key of all other processes.

Key Exchange. The key exchange procedure generates m secret keys, which are essentially random bit strings of length h, and distributes the corresponding verification keys. These are valid for m phases of the k-consensus protocol. If m is equal to or larger than the number of phases required to reach consensus, then the key exchange procedure only needs to be executed once, at the beginning of the k-consensus protocol. Potentially, this scheme can be further optimized so that a single key exchange can span multiple instances of the k-consensus. Nevertheless, for clarity purposes, we describe the scheme assuming only a single instance.

For each process p_i, the key exchange $e \geq 1$ consists of the following steps. Process p_i generates a two-dimensional array SK_i of secret keys, such that each element $SK_i[\phi][v]$ is a random bit string of length h, with $(e-1)m+1 \leq \phi \leq em$ and $v \in \{0, 1, \perp\}$ [3]. It then creates an equivalent two-dimensional array VK_i of verification keys, such that each element $VK_i[\phi][v] = H(SK_i[\phi][v])$. Finally, the verification keys array VK_i is signed using the trapdoor one-way function F and the private key pr_i, and then both the VK_i and the signature are disseminated to the other processes using an out-of-band reliable channel.

When VK_i arrives to a process, the correctness of the keys is confirmed by verifying the signature with the public key of p_i, and then the array is stored for future use. For efficiency purposes, the first VK_i array can be distributed offline along with the public keys. Subsequent arrays may be transmitted during idle periods of the system such that interference with normal execution is kept to a minimum.

Message Authentication. For any phase ϕ, a message $\langle i, \phi, v, status \rangle$ broadcast by process p_i is authenticated by attaching $SK_i[\phi][v]$. When a process p_j receives the message, it applies the hash function to $SK_i[\phi][v]$ and verifies if $H(SK_i[\phi][v])$ is equal to $VK_i[\phi][v]$. If they are equal, then by the properties of cryptographic hash functions ϕ and v originated at p_i.

6.2. Semantic Validation

The semantic validation ensures that the values carried by the three states variables within a message are congruent with the execution of the algorithm. For example, if, at phase $\phi = 1$, every correct process broadcasts the same value 0, then it is not possible for a process that is executing the protocol to send a proposal value of 1 at phase $\phi + 1$. Therefore, if such proposal arrives, then it must have been sent by a Byzantine process, and it can be discarded without impacting the protocol. In practice, this validation mechanism restricts the way that Byzantine processes may lie.

[3] In practice, $SK_i[\phi][\perp]$ only needs to be generated if $\phi \pmod 3 = 0$ because $\perp$ is an acceptable proposal value only in such phases.

There are two ways for the congruency of messages to be verified: one is *implicit* and the other is *explicit*. The implicit way is based on whenever a process receives a message, it sees if enough messages have arrived to justify the values carried by the message just received. For example, if a process has in set V_i more than $\frac{n+f}{2}$ messages with phase ϕ, then, for any message of the form $\langle *, \phi + 1, *, * \rangle$, its phase value is implicitly valid.

The explicit way is based on broadcasting, along with the message, the previous messages that justify the values of the state variables. For example, a message with phase $\phi+1$ can be justified by having appended more than $\frac{n+f}{2}$ messages of the form $\langle *, \phi, *, * \rangle$ (and, naturally, the appended messages must also pass the validity checks).

Our current implementation of the algorithm resorts to both techniques. First, a process tries an implicit validation, which is optimistic by nature, and is much more efficient because messages are allowed to be kept small. However, if, for the following clock tick, a process is forced to broadcast the same message, then explicit validation is employed by appending the justifying messages.

Each of the state variables carried by a message are validated independently. A message passes this validation test if all three variables pass in their individual test. The messages required to validate each variable may sometimes overlap. Next, we explain in more detail how to perform the validations.

Phase value. The phase value ϕ of a message of the form $\langle *, \phi, *, * \rangle$ requires more than $\frac{n+f}{2}$ messages of the form $\langle *, \phi - 1, *, * \rangle$ to be considered valid.

Proposal value. The validation of the proposal value varies according to the phase carried in the message. Messages with phase value $\phi = 1$ are the only that do not require validation and are immediately accepted.

- *Messages with phase* $\phi \pmod 3 = 2$: The proposal value v is valid if there are more than $(\frac{n+f}{2})/2$ messages with phase $\phi - 1$ and proposal value v.

- *Messages with phase* $\phi \pmod 3 = 0$: If the proposal value is $v \in \{0, 1\}$, then it requires more than $\frac{n+f}{2}$ messages with phase $\phi - 1$ and proposal value v. If the proposal value is $\perp$, then it requires more than $(\frac{n+f}{2})/2$ messages of the form $\langle *, \phi-2, 0, * \rangle$ and more than $(\frac{n+f}{2})/2$ messages of the form $\langle *, \phi - 2, 1, * \rangle$.

- *Messages with phase* $\phi \pmod 3 = 1$: The validity of proposal value v in these messages depends if it was obtained deterministically (Line 33) or randomly (Line 35). If obtained deterministically, it requires more than

$\frac{n+f}{2}$ messages of the form $\langle *, \phi - 2, v, * \rangle$. If set randomly, then it requires more than $\frac{n+f}{2}$ messages of the form $\langle *, \phi - 1, \perp, * \rangle$.

Status value. For the $status$ variable, any message with phase $\phi \leq 3$ must necessarily carry value $undecided$ because no process can decide prior to phase 3. For messages with $\phi > 3$, a $status = decided$ (and value v) requires more than $\frac{n+f}{2}$ messages of the form $\langle *, \phi, v, * \rangle$ where $\phi \pmod 3 = 0$. A $status = undecided$ requires more than $(\frac{n+f}{2})/2$ messages of the form $\langle *, \phi', 0, * \rangle$ and more than $(\frac{n+f}{2})/2$ messages of the form $\langle *, \phi', 1, * \rangle$, where ϕ' must be the highest $\phi' \pmod 3 = 2$ lower than ϕ.

7. Performance Evaluation

This section compares the performance of the Turquois protocol with the intrusion-tolerant binary consensus protocols of Bracha [7] and Cachin et al. (named ABBA [8]) in 802.11b wireless ad hoc networks. Like Turquois, both are leader-free randomized protocols that achieve optimal resilience in terms of Byzantine processes. Unlike our protocol, they were not designed with a wireless environment in mind, and employ the typical intrusion-tolerant asynchronous model with reliable point-to-point links.

The protocol of Bracha does not resort to any kind of cryptographic operations, apart from a computationally efficient hash function to authenticate the point-to-point channels, but requires many message exchanges (in complexity order of $O(n^3)$), and the expected worst-case number of rounds to terminate is $O(2^n)$. The ABBA protocol, on the other hand, has message complexity of $O(n^2)$ and terminates in a constant number of steps (at most two rounds of three steps each), but relies heavily on expensive public-key cryptography.

7.1. Testbed and Implementation

The experiments were carried out on the Emulab testbed [29]. A total of 16 nodes were used, each one with the following hardware characteristics: Pentium III processor, 600 MHz of clock speed, 256 MB of RAM, and 802.11 a/b/g D-Link DWL-AG530 WLAN interface card. The operating system was the Fedora Core 4 Linux with kernel version 2.6.18.6. The nodes were located on the same physical cluster and were, at most, a few meters distant from each other.

All the protocols were implemented in C. In Turquois, processes communicate using UDP broadcast. A local clock tick is triggered if one of the following conditions is true: (1) 10 ms have passed since the last broadcast, or (2) the phase value was changed. In both Bracha's and ABBA, the processes use TCP to communicate because of their requirement of reliable point-to-point links. Bracha's protocol requires authenticated channels. To this end, we use the IPSec Authentication Header with security associations being established between every pair of nodes before the execution of the protocol. Both Turquois and ABBA employ their own authentication mechanisms. For these protocols, the cryptographic keys were generated and distributed before the execution of the protocols.

7.2. Methodology

The performance metric utilized in the experiments is the *latency*. This metric is always relative to a particular process p_i, and it is denoted as the interval of time between the moment p_i proposes a value to a consensus execution, and the moment p_i decides.

The average latency for the whole set of processes is obtained in the following manner. A signaling machine, which does not participate in the execution of the protocols, is selected to coordinate the experiment. It broadcasts a 1-byte UDP message to the n processes involved in the experiment. When a process receives such a message, it starts a consensus execution. Processes record the latency value as described above, and send a 1-byte UDP message to the signaling machine indicating the termination of the execution of the protocol. The signaling machine, upon receiving n such messages, waits five seconds, and recommences the procedure. The *average latency* is obtained by repeating this procedure 50 times, and then by averaging the latencies collected by all processes. The confidence interval for the average latency is calculated for a confidence level of 95%.

The experiments were carried out for combinations of group size, proposal distribution, and fault load. The group size defines the number of processes in the system. In our experiments, the values are 4, 7, 10, 13, and 16 processes. The proposal distribution defines the initial values to be proposed by the processes. In the *unanimous* proposal distribution all processes propose the same initial value 1. In the *divergent* distribution processes with an odd process identifier propose 1, while the others propose 0. The *fault load* defines the type of faults that are injected in the system. In the *failure-free* fault load, all processes behave correctly. The *fail-stop* fault load makes $f = \lfloor \frac{n-1}{3} \rfloor$ processes crash before the measurements are initiated. In the *Byzantine* fault load, $f = \lfloor \frac{n-1}{3} \rfloor$ processes try to keep the correct processes from reaching a decision by attacking the execution of the protocol. This is accomplished as follows. In both Bracha's and Turquois, a Byzantine process in phase 1 and 2 proposes the opposite value that it would propose if it were behaving correctly, and in phase 3 it proposes the default value $\perp$. This strategy is followed even if messages are potentially considered invalid. In ABBA, since the protocol

terminates in a constant number of steps, a Byzantine process does not have much room to delay the execution of the protocol by proposing incorrect values. Instead, it transmits messages with invalid signatures and justifications in order to force extra computations at the correct processes. Finally, the value of the parameter k in Turquois is set to $k = n - f$ in all fault loads, with $f = \lfloor \frac{n-1}{3} \rfloor$.

7.3. Results

Failure-free fault load. Table 1 presents the average latency for every tested combination of group size and proposal distribution, in executions without process failures. By observing the results, it becomes apparent that Turquois performs significantly better than the other two protocols. The difference becomes wider as the number of processes increases, exceeding an order of magnitude in some cases.

The performance of Turquois stems naturally from its design. Two fundamental reasons contribute to its efficiency. First, the use of UDP broadcast takes full advantage of the shared communication medium. This was only possible because the protocol is able to tolerate dynamic transmission faults. Second, the use of a novel hash-based signature scheme for message validation allows for computational efficiency. The impact of these features is clearly reflected in the results.

Bracha's protocol is the worst contender, showing serious performance degradation due to the $O(n^3)$ message complexity. In addition to being a shared medium, wireless ad hoc networks are restricted in their speed and capacity, and, therefore, a higher number of message transmissions is bound to have a severe cost. The ABBA protocol performs better than Bracha's, but still much worse than Turquois. Despite its $O(n^2)$ message complexity, the fact that, like Bracha's, it still requires the use of TCP channels combined with heavy cryptography proves to be too much of a burden.

The relative difference between proposal distributions was approximately the same across all protocols, with the latency roughly doubling from an unanimous to a divergent proposal distribution. The reason for this is that when processes propose different values, the protocols usually need to execute for an additional cycle of steps. For example, in Turquois, processes decide by the end of phase 3 with unanimous proposals, but with divergent proposals they typically decide by the end of phase 6. Under the divergent scenario, the first cycle of steps is usually not enough for processes to decide, but is sufficient for a significant number of them to converge into the same proposal value, which leads to a decision by the end of the following cycle.

Fail-stop fault load. Table 2 shows the performance of the protocols when $f = \lfloor \frac{n-1}{3} \rfloor$ processes crash before the execution of the protocols begins. Two observations are clear from these results. First, for all three protocols, there is practically no difference between the two proposal distributions. Since f processes crash, for every group size tested, exactly $n - f = \lfloor \frac{n+f}{2} \rfloor + 1$ processes are left in the system. This means that, as the processes make progress, they necessarily have to receive the same set of messages. Thus, never diverging in their proposal values after the first phase.

The second observation is that, for the unanimous proposal distribution, in most cases the performance of the protocols is worse in the fail-stop scenario than in the fault-free experiments. At a first glance this result seems counterintuitive because when some processes crash there is less contention on the network and, in principle, the protocols can run faster. The problem is that protocols become more sensitive to message loss when only $n-f$ processes are present in the system. More retransmissions are needed to ensure that processes receive enough messages to make progress. Turquois is particularly sensitive to this fact. There are two reasons that explain this: (1) since Turquois uses UDP broadcast, a single collision can result in up to $n - 1$ processes not receiving a message, while in the protocols that employ TCP one collision results in just one process not receiving the message; (2) furthermore, the timeout mechanism in the current implementation of Turquois is crude when comparing to the sophistication of TCP, and is not adaptable to network conditions nor to the number of processes involved in the communication. This also explains its proportionally wider confidence interval. An optimization of the retransmission mechanism could significantly improve the performance of Turquois in these scenarios. Nevertheless, Turquois still performs significantly better than the other two protocols with this fault load.

There are two exceptions to the observation that protocols perform better in the failure-free fault load when compared with the fail-stop fault load. They occur in Bracha's and ABBA when $n = 16$. This indicates that there may be a turning point where the group size becomes more stringent to performance than sensitivity to message loss, although experiments with higher numbers of processes would be necessary to confirm this.

Byzantine fault load. Table 3 shows the performance of the protocols when $f = \lfloor \frac{n-1}{3} \rfloor$ processes act according to a malicious strategy. It is interesting to note that the *relative* difference between the unanimous and divergent proposal distributions is similar to the scenario with no process failures, with the latency very roughly doubling in the divergent distribution. Like in the failure-free scenario, this is due to divergent proposal values forcing processes to execute for extra rounds to reach a decision.

When compared directly to the failure-free scenario, this fault load suffers from a performance degradation that be-

Group Size	Average Latency ± Confidence Interval (ms)					
	Turquois		ABBA		Bracha	
	unanimous	divergent	unanimous	divergent	unanimous	divergent
$n = 4$	14.90 ± 4.74	28.67 ± 9.99	74.70 ± 7.93	135.39 ± 28.04	101.06 ± 8.15	127.39 ± 22.99
$n = 7$	26.85 ± 6.18	54.38 ± 12.20	125.81 ± 6.22	253.66 ± 37.93	552.77 ± 31.36	715.15 ± 112.90
$n = 10$	43.15 ± 10.05	71.75 ± 25.05	277.90 ± 12.47	547.42 ± 81.94	1361.90 ± 33.17	2282.23 ± 315.53
$n = 13$	60.94 ± 14.15	128.07 ± 42.51	693.39 ± 103.45	1722.44 ± 295.05	3459.10 ± 100.34	6276.91 ± 734.11
$n = 16$	87.57 ± 22.34	236.31 ± 77.27	1914.54 ± 283.18	4309.51 ± 750.20	7321.41 ± 110.69	10420.00 ± 2640.11

Table 1. Average latency and confidence interval in a 802.11b network with no process failures (latency in milliseconds and confidence level of 95%).

Group Size	Average Latency ± Confidence Interval (ms)					
	Turquois		ABBA		Bracha	
	unanimous	divergent	unanimous	divergent	unanimous	divergent
$n = 4$	42.26 ± 30.29	43.84 ± 31.27	77.31 ± 9.17	77.88 ± 9.34	99.29 ± 3.05	99.61 ± 3.17
$n = 7$	106.28 ± 37.98	110.18 ± 22.00	183.20 ± 15.96	169.90 ± 6.18	516.26 ± 26.70	519.76 ± 37.63
$n = 10$	168.45 ± 39.46	188.95 ± 35.05	310.97 ± 15.61	335.93 ± 24.09	2488.75 ± 52.53	2619.35 ± 75.43
$n = 13$	375.00 ± 56.03	387.22 ± 60.06	747.56 ± 44.77	771.68 ± 52.71	5992.63 ± 143.00	6267.88 ± 355.51
$n = 16$	395.96 ± 55.11	422.65 ± 82.41	1180.03 ± 109.18	1284.83 ± 103.64	6362.68 ± 136.64	6469.38 ± 159.40

Table 2. Average latency and confidence interval in a 802.11b network with fail-stop process failures (latency in milliseconds and confidence level of 95%).

comes increasingly noticeable with a higher group size, specially with a divergent proposal distribution. The reason for this is that many messages broadcast by Byzantine processes carry values that fail to pass the validation mechanisms of the protocols. The result is that, similarly to the fail-stop scenario, protocols become sensitive to message loss with the added burden of a higher contention (with n processes broadcasting messages). As for Turquois, despite its non-optimized timeout mechanism making it more sensitive to this issue, it still manages to be the faster protocol.

8. Conclusions

The paper presented Turquois, an intrusion-tolerant binary consensus protocol specifically designed for wireless ad-hoc networks. Its design takes into account the typically constrained resources of wireless ad-hoc environments, while aiming for optimal resilience parameters. The protocol tolerates $f < \frac{n}{3}$ Byzantine processes. Furthermore, it assumes communication to be inherently unreliable by incorporating the *communication failure model* [25]. *Safety* is maintained despite unrestricted message omissions, and *liveness* is ensured in rounds where the number of omissions is bounded by $\sigma \leq \lceil \frac{n-t}{2} \rceil (n-k-t) + k - 2$, where k is the number of processes required to decide, and $t \leq f$ is the number of processes that are actually faulty. The timing assumptions are also very weak, requiring only a local timeout on each process to ensure these keep sending messages.

The key to its performance was the decision to assume unreliable communication, which allows the protocol to take full advantage of the broadcasting medium, where the cost of transmitting a message to multiple nodes can be just the same of sending it to a single one. Furthermore, the protocol avoids the use of public-key cryptography during its normal operation in order to preserve the computational power of mobile nodes, which is usually limited. The protocol was subject to a comparative performance evaluation against two well-known intrusion-tolerant consensus protocols. The results showed that, regardless of the type of faults present in the system, Turquois significantly outperforms the other protocols, in particular as the number of processes in the system increases.

References

[1] N. Badache, M. Hurfin, and R. Macedo. Solving the consensus problem in a mobile environment. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference*, pages 29–35, 1999.

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

[3] A. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Veríssimo. The CRUTIAL way of critical infrastructure protection. *IEEE Security and Privacy*, 6(6):44–51, 2008.

[4] V. Bhandari and N. Vaidya. On reliable broadcast in a radio network. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 138–147, 2005.

[5] M. Biely, J. Widder, B. Charron-Bost, A. Gaillard, M. Hutle, and A. Schiper. Tolerating corrupted communication. In

Group Size	Average Latency ± Confidence Interval (ms)					
	Turquois		ABBA		Bracha	
	unanimous	divergent	unanimous	divergent	unanimous	divergent
$n = 4$	44.74 ± 30.16	80.18 ± 33.93	87.65 ± 22.38	197.78 ± 25.25	111.16 ± 6.99	248.66 ± 38.80
$n = 7$	96.20 ± 37.88	186.74 ± 60.54	198.69 ± 17.72	361.53 ± 48.41	619.09 ± 23.40	1634.17 ± 236.21
$n = 10$	145.22 ± 23.21	288.94 ± 64.04	481.83 ± 31.10	1137.94 ± 37.78	2216.42 ± 54.17	5633.47 ± 668.64
$n = 13$	386.39 ± 38.57	719.79 ± 72.57	1573.46 ± 110.70	3276.53 ± 211.76	5445.93 ± 114.10	12656.41 ± 1572.59
$n = 16$	590.95 ± 76.14	904.27 ± 83.48	2940.68 ± 426.93	6045.06 ± 533.52	7698.29 ± 180.10	20412.36 ± 2271.55

Table 3. Average latency and confidence interval in a 802.11b network with Byzantine process failures (latency in milliseconds and confidence level of 95%).

Proceedings of the 26th ACM Symposium on Principles of Distributed Computing, pages 244–253, 2007.

[6] F. Borran, R. Prakash, and A. Schiper. Extending Paxos/LastVoting with an adequate communication layer for wireless ad hoc networks. In *Proceedings of the 27th IEEE International Symposium on Reliable Distributed Systems*, pages 227–236, 2008.

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

[8] 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.

[9] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[10] B. Charron-Bost and A. Schiper. The heard-of model: Computing in distributed systems with benign failures. Technical Report LSR-REPORT-2007-001, EPFL, 2007.

[11] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, 2005.

[12] V. Drabkin, R. Friedman, and M. Segal. Efficient byzantine broadcast in wireless ad-hoc networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 160–169, 2005.

[13] 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.

[14] C. Koo. Broadcast in radio networks tolerating Byzantine adversarial behavior. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 275–282, 2004.

[15] C.-Y. Koo, V. Bhandari, J. Katz, and N. H. Vaidya. Reliable broadcast in radio networks: the bounded collision case. In *Proceedings of the 25th annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM, 2006.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[17] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[18] H. Moniz, N. F. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks (extended version). Technical Report DI/FCUL TR-10-02, Department of Computer Science, University of Lisbon, 2010.

[19] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo. Randomization can be a healer: Consensus with dynamic omission failures. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 63–77, 2009.

[20] A. Pelc and D. Peleg. Broadcasting with locally bounded byzantine faults. *Information Processing Letters*, 93(3):109–115, 2005.

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

[22] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938, pages 99–110. Springer-Verlag, 1995.

[23] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

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

[25] N. Santoro and P. Widmeyer. Time is not a healer. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science*, pages 304–313, 1989.

[26] 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.

[27] H. Seba, N. Badache, and A. Bouabdallah. Solving the consensus problem in a dynamic group: an approach suitable for a mobile environment. In *Proceedings of the 7th IEEE International Symposium on Computers and Communications*, pages 327–332, 2002.

[28] E. Vollset and P. D. Ezhilchelvan. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in MANETs. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 166–175, 2005.

[29] 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*, pages 255–270, 2002.

[30] W. Wu, J. Cao, J. Yang, and M. Raynal. Design and performance evaluation of efficient consensus protocols for mobile ad hoc networks. *IEEE Transactions on Computers*, 56(8):1055–1070, August 2007.