Experimental Comparison of Local and Shared Coin Randomized Consensus Protocols*

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

Abstract

The paper presents a comparative performance study of the two main classes of randomized binary consensus protocols: a local coin protocol, with an expected high communication complexity and cheap symmetric cryptography, and a shared coin protocol, with an expected low communication complexity and expensive asymmetric cryptography. The experimental evaluation was conducted on a LAN environment, by varying several system parameters, such as the fault types and number of processes. The analysis shows that there is a significant gap between the theoretical and the practical performance results of these protocols, and provides an important insight into what actually happens during their execution.

1. Introduction

Consensus is a cornerstone problem in distributed systems. It is related to several important fault-tolerant services, including the state-machine replication, atomic commitment, group membership and total order broadcast [10]. Basically it can be described as follows: each process starts with a private value that is used as a proposal for consensus, and then, when a conclusion is reached, all processes obtain the same decision on one of the proposed values. Behind this simple definition, consensus ends up being a complex problem, specially if one starts to consider potential failures of the individual components of the system. In fact, consensus was proven impossible to solve deterministically in asynchronous systems –i.e., systems with no bounds on the computation and communication times– where a single process crash can occur (the FLP result) [12].

The consensus problem is even more stringent if not only crash but also arbitrary failures can take place. Dealing with this class of failures, often dubbed Byzantine failures, has a practical interest because they include malicious actions of human origin, such as attacks and intrusions. Since these attacks can be directed against the synchrony properties of the system, for example by introducing artificial delays with denial of service attacks, it is usually advisable to assume an asynchronous model when this kind of failures are being considered. The consequence of this decision is that the system becomes bound to the FLP result. Nevertheless, recently there has been a significant interest in developing solutions for systems with these characteristics because they are capable of automatically tolerate intrusions, which contributes to an overall improved security [13, 19].

Therefore, in practice, solving consensus in a weak system model in which FLP applies, requires something that is usually called *to circumvent FLP*. Throughout the years several techniques have been used for this purpose. All these techniques, with the exception of randomization, require the extension of the basic system model with additional time assumptions, either explicitly (e.g., partial synchrony models [11]) or implicitly (e.g., failure detectors [7] and wormholes [9]). Randomization, on the other hand, solves consensus probabilistically rather than deterministically, and allows the system to preserve a completely asynchronous nature. This technique, however, has often been considered too inefficient, mainly because it leads to protocols with large expected time and message complexities. Only recently a randomized consensus protocol that runs in a reduced number of rounds has been proposed [4] and other algorithms have been shown to be efficient in practical settings with realistic faultloads [16].

Virtually all randomized consensus protocols are based on a random operation, *tossing a coin*, which returns values 0 or 1 with equal probability. These protocols can be divided in two classes depending on how the tossing operation is performed: there are those that use a *local coin* mechanism in each process (starting with Ben-Or's work [1]), and those based on a *shared coin* that gives the same values to all processes (initiated with Rabin's work [17]). Typically, local coin protocols are simpler but terminate in an expected

*This work was partially supported by the EU through NoE IST-4-026764-NOE (RESIST) and project IST-4-027513-STP (CRUTIAL), and by the FCT through project POSI/EIA/60334/2004 (RITAS) and the Large-Scale Informatic Systems Laboratory (LASIGE).

exponential number of rounds, while shared coin protocols require an additional coin sharing scheme but can terminate in an expected constant number of rounds [6, 4].

In this paper, we compare the performance of the two classes of randomized protocols in a LAN setting, in order to get a better understanding about the trade-offs involved in their construction. To our knowledge, this is the first time such analysis in being made. To conduct this investigation, we have selected one protocol from each class, and studied their behavior under different conditions, such as distinct faultloads, using the latency and throughput metrics.

For the shared coin class, the chosen protocol was the most efficient protocol of the kind that we are aware of, ABBA [4]. This protocol uses an interesting and innovative combination of cryptographic primitives, like threshold signatures and a threshold coin-tossing scheme, which makes it have a very good time complexity – it reaches decision in one or two rounds with a high probability. For the local coin class was selected the Bracha's classical protocol [3]. This protocol resorts to almost no cryptographic operations, but potentially runs in an exponential number of rounds. Nevertheless, it has been shown to be efficient in practice, when used as part of an intrusion-tolerant stack of protocols [16].

Both protocols solve *binary consensus*, i.e., the values proposed and decided are binary digits, 0 or 1. This is a natural choice since most randomized consensus protocols are binary. Randomized multi-valued consensus and other variants are usually implemented on top of binary consensus protocols (see, e.g., [5, 10]).

The experimental evaluation lead to several interesting conclusions. The local coin protocol was the fastest in all experiments. However, the shared coin protocol is apparently more scalable, since its performance degraded less when the number of processes was increased. The local coin protocol was also more sensitive to a network bandwidth decrease than the other, which indicates that it should perform worse in environments where the bandwidth is limited (e.g., a WAN). The average number of rounds executed in practice was close to 1 in both protocols, although theoretically the local coin protocol runs in an exponential number of rounds.

The rest of the paper of organized as follows. Section 2 describes the basic system model. Section 3 formally defines the binary consensus problem. Both protocols are described in Section 4. Section 5 presents a detailed comparative performance evaluation between the two protocols. The related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2 Basic System Model

The basic system model describes the properties that are common to both algorithms. Specific characteristics of the algorithms are described in their respective sections as extensions to this model.

The system is composed by a set of n processes $P = \{p_0, p_1, ...p_{n-1}\}$. The processes are said to be *correct* if they follow the protocol until termination (i.e., they do not fail). Processes that fail are said to be *corrupt*. A maximum of $f = \lfloor \frac{n-1}{3} \rfloor$ processes can be corrupt during the lifetime of the system. There are no constrains on the actions taken by corrupt processes – they can, for instance, stop executing, omit messages, or send invalid messages, either alone or in collusion with other corrupt processes. This class of unconstrained faults is usually called *arbitrary* or *Byzantine*. The system is asynchronous meaning that no assumptions are made about the bounds on processing times or communication delays. It is assumed that the communication channels are unreliable, which indicates, for instance, that messages can be lost or (maliciously) modified while in transit in the network. We assume the adversary cannot break the cryptography employed in the protocols.

3 The Binary Consensus Problem

In the binary consensus problem, a set of processes p_i proposes some initial value $v_i \in \{0, 1\}$ and then they must decide on a common value. Since we are considering the randomized model, the termination of the protocol in only guaranteed in a probabilistic way. More formally, the binary consensus problem is specified with the following properties:

Agreement If all correct processes propose the same value v, then any correct process that decides, decides v.

Validity No two correct processes decide differently.

Termination All correct processes eventually decide with probability 1.

4. The Binary Consensus Protocols

This section gives an overview of the two protocols that were studied (more details can be obtained in the original papers [3, 4]). Bracha's protocol has a high time and communication complexity but (almost) entirely avoids the use of cryptography, while the ABBA protocol has a low time and communication complexity but uses several asymmetric (or public-key) cryptography primitives. Both protocols require some extensions to the basic system model which are discussed in the corresponding subsections. Additionally, they also employ message validation techniques whose purpose is to force corrupt processes to broadcast messages with values that are congruent with the values proposed by the other processes (or otherwise, the malicious values are detected and ignored by the correct processes).

4.1. Bracha's Local Coin Protocol

Bracha's binary consensus protocol exchanges $O(n^3)$ point-to-point messages per round and the expected number of rounds until termination is 2^{n-f} under the *strong adversary* model[1]. The algorithm itself does not use any kind of cryptographic operations, albeit its dependence on a reliable communication channel implies the use of a relatively inexpensive cryptographic hash function.

Communication Primitives. The protocol needs some extensions to the basic system model to provide the binary consensus properties. More specifically, it requires a *reliable channel* abstraction, and on top of this abstraction a *reliable broadcast* primitive.

The reliable channel abstraction offers point-to-point communication between any pair of correct processes with the *reliability* and *integrity* properties. Reliability means that messages are eventually received, and integrity says that messages are delivered without modifications (i.e., messages with changes are detected and removed). In practical terms, these properties can be enforced using retransmissions and Message Authentication Codes (MACs). A MAC is a cryptographic checksum which can be calculated with a hash function and a shared key [15]. Therefore, it is assumed that every pair of processes (p_i, p_j) share a secret key k_{ij}. The way these keys are given to the processes is out of the scope of the protocol, but it may require some kind of trusted dealer or key distribution protocol. Nevertheless, since this task is performed during initialization, it does not affect performance during the execution of the protocol. Standard Internet protocols can be employed to implement the reliable channels. In the experiments, reliability was achieved with TCP and integrity with the IPSec Authentication Header protocol [14].

The reliable broadcast primitive keeps corrupt processes from broadcasting conflicting values to different processes. It ensures that: (1) all correct processes deliver the same messages, and (2) if the sender is correct then the message is delivered. This primitive can be implemented in different ways. In our case, we have used the following protocol [3]. The sender starts by broadcasting a (INITIAL, m) message to all processes. Upon receiving this message a process transmits a (ECHO, m) message to all processes. It then waits for at least $\lfloor \frac{n+f}{2} \rfloor + 1$ (ECHO, m) messages or $f + 1$ (READY, m) messages, and then it sends a (READY, m) message to all processes. Finally, a process waits for $2f + 1$ (READY, m) messages to deliver m.

[1] In the strong adversary model it is assumed that the adversary completely controls the network scheduling, having the power to decide the timing and the order in which the messages are delivered to the processes.

Protocol Execution. The protocol proceeds in 3-step rounds, running as many rounds as necessary for a decision to be reached. Each process p_i executes a round as follows:

Step 1 Reliable broadcast the initial proposal value. Wait for $n - f$ *valid* messages for this step (the meaning of *valid* is explained in the next section). Set the new proposal value to reflect the majority of the received values. If all the $n-f$ messages have the same value v, then decide, but continue the execution of the protocol to allow the other processes also to finish.

Step 2 Reliable broadcast the proposal value. Wait for $n - f$ *valid* messages for this step. The new proposal value is set to $v \in \{0, 1\}$ if more than $n/2$ of the received messages have the same value v. Otherwise, the new proposal value is set to a default value $\perp$.

Step 3 Reliable broadcast the proposal value. Wait for $n - f$ *valid* messages for this step. If at least $2f + 1$ messages have the same value $v \neq \perp$, then the process decides v (if it had not decided previously). Otherwise, if at least $f + 1$ have the same value $v \neq \perp$, then the process sets the new proposal value to v and a new round is initiated. If none of the previous conditions apply, then the process sets the new proposal value to a random bit with value 1 or 0, each with probability $\frac{1}{2}$, and a new round is initiated.

Message Validation. A message received in the first step of the first round is always considered *valid*. A message received in any other step k, for $k > 1$, is *valid* if its value is congruent with any subset of $n - f$ values accepted at step $k - 1$. Suppose that process p_i receives $n - f$ messages at step 1, where the majority has value 1. Then at step 2, it receives a message with value 0 from process p_j. Remember that the message a process p_j broadcasts at step 2 is the majority value of the messages it received at step 1. That message cannot be considered *valid* by p_i since value 0 could never be derived by a correct process p_j that received the same $n - f$ messages at step 1 as process p_i. If process p_j is correct, then p_i will eventually receive the necessary messages for step 1, which will enable it to form a subset of $n - f$ messages that validate the message with value 0.

4.2. The ABBA Shared Coin Protocol

The ABBA binary consensus protocol exchanges $O(n^2)$ point-to-point messages per round and reaches a decision in 1 or 2 rounds with high probability. The protocol makes extensive use of asymmetric cryptography to ensure the correctness of the execution.

Communication and Cryptographic Primitives. The protocol needs the following extensions to the basic system model: reliable channels for point-to-point communication, and two cryptographic primitives – dual threshold signatures and a threshold coin-tossing scheme.

The reliable channels in the ABBA protocol only need to provide the reliability property (i.e., that messages are eventually received) between every pair of correct processes[2]. The integrity of the messages is guaranteed by the use of public-key signatures inside the protocol itself. Therefore, the TCP protocol can be employed in the implementation of these channels.

An (n, k, f) *dual-threshold signature scheme* is a technique where n processes, from which up to f can be corrupt, hold *shares* of a private key. The processes can generate *shares of signatures* on particular messages, and k of such shares are both necessary and sufficient to assemble a valid signature. Every process has the ability to individually verify every generated share and the assembled signature. In practice, this scheme can be (and was) implemented using a vector of RSA signatures [4].

An (n, k, f) *dual-threshold coin-tossing scheme* is also a technique where there are n processes and at most f of them may be corrupt. Processes hold shares of an unpredictable function F that maps the coin name C to a binary value $F(C) \in \{0, 1\}$. The processes can generate shares of the coin and k of those shares are both necessary and sufficient to assemble the function F. The implemented threshold coin-tossing scheme is the Diffie-Hellman based solution of Cachin et al. [4].

Protocol Execution. Except for the first round where there is an additional communication step, the protocol proceeds in rounds of three steps each. Let cid be a unique identifier for each execution of the protocol. Every process p_i executes the protocol as follows:

Step 0 (first round only) Broadcast a *pre-process* message containing the initial proposal value v_i along with an $(n, f + 1, f)$-signature share on the message *(cid, pre-process, v_i)*. Wait for $2f + 1$ *valid* pre-process messages.

Step 1 If in round $r = 1$, the new proposal value v_i is the majority value of the received *pre-process* messages. If in round $r > 1$, wait for $n - f$ *coin* messages and let $v_i = b$ if there was a *main-vote* in round $r - 1$ for $b \in \{0, 1\}$. Otherwise, let $v_i = F(C)$, where $C = (cid, r)$. Broadcast a *pre-vote* with value v_i along with an $(n, n - f, f)$-signature share on the message *(cid, pre-vote, r, v_i)*.

Step 2 Wait for $n - f$ *valid* pre-votes. If there were $n - f$ pre-votes for $b \in \{0, 1\}$, then set $v_i = b$. Otherwise, set $v_i = abstain$. Broadcast a *main-vote* with value v_i along with an $(n, n - f, f)$-signature share on the message *(cid, main-vote, r, v_i)*.

Step 3 Wait for $n - f$ *valid* main-votes. If there were $n - f$ main-votes for b, then decide b and continue for one more round up to step 2. Otherwise, generate a share of the coin with name $C = (cid, r)$. Broadcast the *coin share* in a coin message, and proceed for round $r = r + 1$.

Message Validation. In round $r = 1$, a pre-vote for value b is valid when accompanied by an $(n, f + 1, f)$-threshold signature on the message *(cid, pre-process, b)*. In round $r > 1$, a pre-vote for value b is valid when accompanied by either an $(n, n - f, f)$-threshold signature on the message *(cid, pre-vote, r-1, b)* (hard pre-vote), or an $(n, n - f, f)$-threshold signature on the message *(cid, main-vote, r-1, abstain)* (soft pre-vote). A hard pre-vote is cast when there was a main-vote for either 0 or 1 in round $r - 1$. A soft pre-vote is cast when all the main-votes in round $r - 1$ were $abstain$. The soft pre-vote value is $F(cid, r - 1)$.

A main-vote for value $abstain$ in round r is valid when it is accompanied by either the validations of two conflicting round r pre-votes (i.e., one pre-vote for value 0, and another for 1). A main-vote for value $b \in \{0, 1\}$ in round r is valid when it is accompanied by an $(n, n - f, f)$-threshold signature on the message $(cid, pre - vote, r, b)$.

Besides these validations, all received signature shares also need to be verified to accept the corresponding messages. This also includes the coin shares generated in step 3. They need to be verified before being assembled into a coin function $F(C)$.

5. Performance Evaluation

This section provides a thorough performance-wise analysis of the local coin (LCP) and shared coin (SCP) protocols on a local area network. The algorithms were evaluated under several different environmental conditions that can be adjusted as system parameters.

The experiments were conducted on a testbed consisting of 11 Dell PowerEdge 850 computers. The characteristics of the machines are the same: a single Pentium 4 CPU with 2.8 GHz of clock speed, and 2Gb of RAM. The machines were connected by a Dell PowerConnect 2724 network switch with 10/100/1000 Mbps bandwidth capacity. The operating system was Linux, with kernel version 2.6.11. The protocols were implemented using the C language and were compiled with gcc.

[2]Even tough "Bracha's reliable channels" have one more property than the "ABBA reliable channels", we decided to call them with the same name in order to avoid an extra channel qualifier, and therefore to keep the presentation as simple as possible.

5.1. Performance Metrics and System Parameters

The metrics are the set of criteria used to compare the performance of the protocols. The system parameters are the configurable variables of the system that define specific execution environments.

The two main performance metrics utilized in most experiments were the *latency* (L) and the *maximum throughput* (T_{max}). Latency is always relative to a particular process p_i, and 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 consensus value. More than latency, however, we evaluate the protocols in terms of the *burst latency* (L_{burst}). Given a burst of k concurrent consensus executions, the burst latency is the interval of time between the moment p_i proposes the first value and the moment it decides the k^{th} value. The throughput is the number of decisions per second obtained for a burst of a given size k. It is calculated by dividing the burst size k by the burst latency L_{burst}. The maximum throughput T_{max} is the value at which the throughput stabilizes (i.e., does not change with increasing burst sizes).

An additional metric is used in some experiments, the *number of rounds* until decision. This metric, however, should be taken with a grain of salt since a higher number of rounds does not necessarily mean that a protocol is slower. An algorithm may need a higher number of rounds to achieve termination, but these steps may be executed faster.

The system parameters selected for the experiments were the faultload, distribution of process proposals, group size, network bandwidth, and cryptography.

The *faultload* defines the types of faults that are injected in the system during its execution. In the *failure-free* faultload, all processes behave correctly. The *fail-stop* faultload makes f processes crash before the measurements are taken. In the *Byzantine* faultload, f processes try to keep the correct processes from reaching a decision by attacking the protocol execution. This is accomplished as follows. In the LCP coin protocol, a Byzantine process in steps 1 and 2 always proposes the opposite value that it would propose if it were behaving correctly, and in step 3 always proposes the default value $\perp$. In the SCP, since a Byzantine process has no possibility of proposing an invalid value without detection (because of the employed asymmetric cryptography), it transmits messages with invalid signatures and justifications in order to force extra computation in the correct processes.

The *proposal distribution* defines the initial values to be proposed by the processes. The *uniform* proposal distribution makes all processes propose the same initial value 1. In the *corrosive* proposal distribution, processes with an odd process identifier propose 1 and the others propose 0. The *random* proposal distribution chooses for the initial proposal of each process a randomly selected value.

The *group size* defines the number of processes n in the system. In our case it can take three values: 4, 7, and 10.

The *network bandwidth* is the amount of data that can be transmitted between every pair of processes in a given period of time. It can take three values: 10 Mb/s, 100 Mb/s, and 1000 Mb/s. The default network bandwidth used in the experiments was 1000 Mb/s.

Cryptography defines the type of cryptography employed by the protocols. The LCP can only either use the IPsec AH protocol (with SHA-1) or no cryptography at all (in this case the protocol correctness is affected but this setting is utilized for the sake of comparative evaluation only). The SCP uses RSA with three possible key sizes: 512, 1024, and 2048 bits. The cryptographic default settings are set to IPSec for the LCP, and 1024-bit RSA keys for the SCP.

5.2. Basic Latency Measurements

The results presented in this subsection are a starting point for the rest of the performance analysis. They show the latency values for both consensus protocols with different group sizes and proposal distributions. The remaining parameters were set to the default values and no faults were injected (i.e., the failure-free faultload).

The measurements were taken the following way: a signaling machine, which does not participate in the protocols, is selected to control the benchmark execution. It broadcasts m 1-byte UDP messages to the n processes involved in the experiment, each one separated by a five second interval (in this case m was set to 100). Whenever one of these messages arrives to a specific process, it executes whatever protocol is relevant for the current experiment (LCP or SCP). Processes record the interval of time denoted by the instant they receive the signal message and the instant they get a decision. This period is the latency value for each process. The *average latency* is obtained by taking the mean value of the sample of measured values.

	Local Coin (μs)			Shared Coin (μs)		
	$n = 4$	$n = 7$	$n = 10$	$n = 4$	$n = 7$	$n = 10$
uniform	824	2187	4132	21590	31315	43633
corrosive	2453	6172	12075	33834	38529	55169
random	2056	5812	11501	24320	36325	49206

Table 1. Average latency in microseconds (μs) for different group sizes and proposal distributions.

The results of this experiment are shown in Table 1. It can be observed that the LCP is very fast, reaching a decision in less than 1 ms with 4 processes and a uniform proposal distribution. The SCP is comparatively slower, despite having a lower communication complexity. In these environmental settings, the lower number of exchanged

messages of the SCP clearly does not compensate for the computationally intensive asymmetric cryptography.

Nevertheless, even tough the LCP is significantly faster than the SCP, its latency grows at a faster rate as the group size increases. While the latency of the LCP roughly doubles at successive larger group sizes, the latency for the SCP grows at roughly 50%. This indicates that SCP could potentially outperform LCP in groups with high numbers of processes.

5.3. Protocol Behavior under Different Faultloads

This section studies the behavior of the protocols when subject to different faultloads. Two system parameters were varied in this experiment: the faultload and the number of processes. The proposal distribution was fixed to random, and the rest of the parameters were set to the default values. The metrics used to assess the performance of the protocols were the latency, throughput, and number of rounds.

The experiment was carried out by having the n processes run several concurrent consensus executions. A signaling machine, which does not participate in the execution of the protocols, sends a 2-byte UDP message to all n processes, containing a number k. When a processes receives this message, it starts k simultaneous consensus executions. Every process measures the interval of time between the instant it receives the signal message, and the instant it reaches the k^{th} decision. This interval of time is the latency of the burst of k consensus executions (L_{burst}). The burst throughput is obtained by dividing the burst size k by the burst latency L_{burst}. For every tested burst size k, the displayed result reflects the average value of 10 executions.

The results for each faultload are presented in three separate pairs of graphs. The first graph of each pair studies the latency and the second the throughput. An analysis on the number of rounds is presented at the end of this subsection.

Failure-free Faultload. The results when there are no faults are shown in Figures 1 and 2, respectively for the LCP and the SCP protocols. Each curve represents a different group size n.

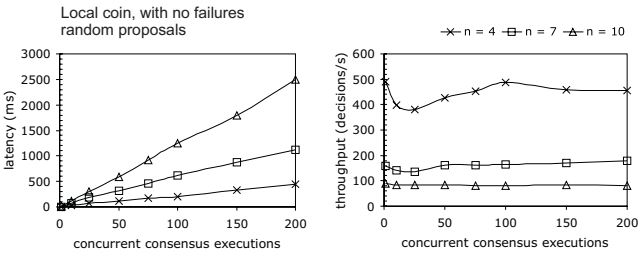

Figure 1. Burst latency and throughput for the LCP with no failures

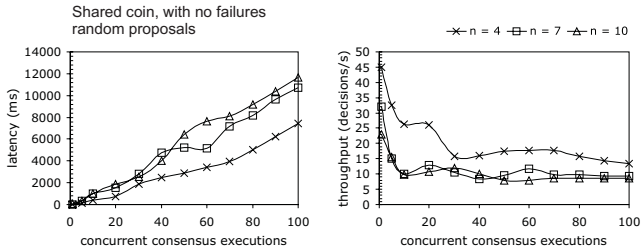

Figure 2. Burst latency and throughput for the SCP with no failures

From the graphs it is possible to observe that the burst latency L_{burst} is linear with the number of concurrent consensus executions. The stabilization points in the throughput curves indicate the maximum throughput T_{max} for each n. It can be seen that the LCP is much faster than the SCP, achieving much lower latency and higher throughput values, irrespective of the group size.

For the LCP, for 200 concurrent consensus executions, L_{burst} has a value of 439 ms with $n = 4$, 1126 ms with $n = 7$, and 2492 ms with $n = 10$. The maximum throughput T_{max} is around 455 decisions/s with $n = 4$, 175 decisions/s with $n = 7$, and 81 decisions/s with $n = 10$.

As for the SCP, for 100 concurrent consensus executions, L_{burst} has a value of 7454 ms with $n = 4$, 10713 ms with $n = 7$, and 11625 ms with $n = 10$. The maximum throughput T_{max} is around 13 decisions/s with $n = 4$, 9 decisions/s when $n = 7$, and 8 decisions/s when $n = 10$.

Fail-stop Faultload. Figures 3 and 4 display the performance of the protocols when there are f crashed processes in the system. Each curve shows the latency and throughput for a different group size n.

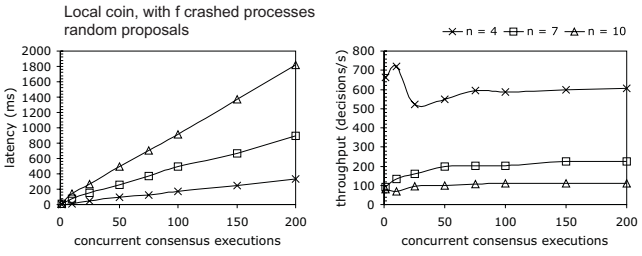

Figure 3. Burst latency and throughput for the LCP with f crashed processes

For the LCP, the performance is noticeably better with f crashed processes than it is in the failure-free scenario. This happens because with fewer processes there is less contention on the network and more bandwidth is available to exchange the messages. This result gives a hint that the performance bottleneck of the LCP is indeed the communication (as opposed to the computation). In more detail, for

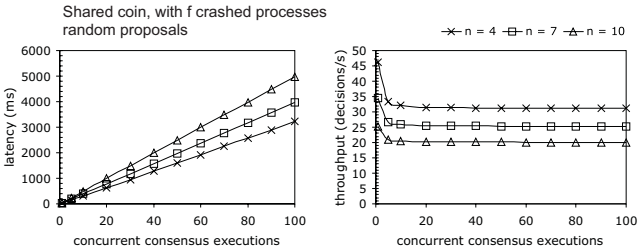

Figure 4. Burst latency and throughput for the SCP with f crashed processes

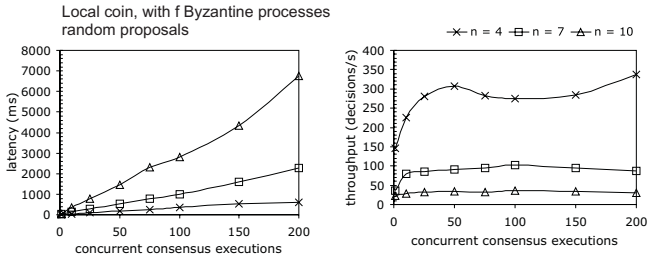

Figure 5. Burst latency and throughput for the LCP with f Byzantine processes

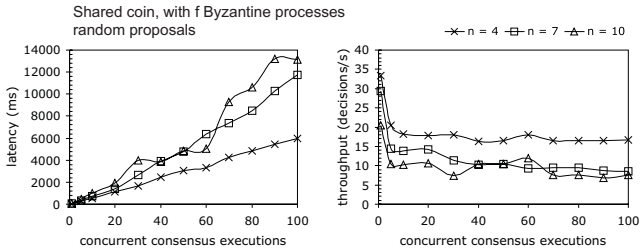

Figure 6. Burst latency and throughput for the SCP with f Byzantine processes

200 concurrent consensus executions, L_{burst} has a value of 329 ms with $n = 4$, 890 ms with $n = 7$, and 1820 ms with $n = 10$. The maximum throughput T_{max} is around 608 decisions/s with $n = 4$, 225 decisions/s with $n = 7$, and 110 decisions/s with $n = 10$.

As for the SCP, the performance results when there are f crashed processes also show a significant improvement over the failure-free scenario. In this case, however, the reduced network contention does not explain entirely what happens. In more detail, for 100 concurrent consensus executions, L_{burst} has a value of 3215 ms with $n = 4$, 3959 ms with $n = 7$, and 4981 ms with $n = 10$. The maximum throughput T_{max} is around 31 decisions/s with $n = 4$, 25 decisions/s when $n = 7$, and 20 decisions/s when $n = 10$.

While this can not be inferred from the graphs, what really speeds up the SCP is the fact that, with only $n - f$ processes proposing values, all the correct processes see exactly the same $n-f$ messages at every protocol step, resulting always in 1-round decisions for all protocol executions (this behavior also happens in the LCP, but its performance is affected to a lesser degree by it). In the failure-free scenario, even with an overwhelming majority of executions reaching decision in only one round, it only takes a single execution needing more than one round to considerably delay the burst latency, hence the performance improvement in the fail-stop case.

Byzantine Faultload. Figures 5 and 6 depict the protocols' performance when there are f processes trying to disrupt their execution. Each curve shows the latency and throughput for a different group size n.

For the LCP, the curves show that the performance is negatively affected by the Byzantine failures. For 200 concurrent consensus executions, L_{burst} has a value of 592 ms with $n = 4$, 2290 ms with $n = 7$, and 6772 ms with $n = 10$. The maximum throughput T_{max} is around 337 decisions/s with $n = 4$, 87 decisions/s with $n = 7$, and 30 decisions/s with $n = 10$. While it is not possible for f malicious processes to prevent correct processes from reaching a decision, it is still possible for them to increase the number of rounds needed to reach a decision. This can be directly linked to the lack of cryptographic verifications on the received messages, which affects the robustness. In practice, this means that the processes usually have to wait for extra messages beyond the $n-f$ threshold because the malicious processes are proposing values that fail the validation. A good example can be found in step 3 of the the protocol. In alternative to $b \in \{0, 1\}$, the processes are allowed to propose an 'undecided' default value $\perp$ for which the verification step is somewhat relaxed (it only needs $n - f - \frac{n}{2}$ *valid* different proposals from the majority value in step 2 to be considered valid by a correct process), which Byzantine processes exploit in order to try to postpone a decision for one extra round.

For the SCP, it is clear from the curves that there is not a noticeable performance penalty in relation to the failure-free scenario. In more detail, for 100 concurrent consensus executions, L_{burst} has a value of 5987 ms with $n = 4$, 11712 ms with $n = 7$, and 13139 ms with $n = 10$. The maximum throughput T_{max} is around 16 decisions/s with $n = 4$, 9 decisions/s when $n = 7$, and 8 decisions/s when $n = 10$. These results are directly related to the extensive use of public-key cryptography which provides superior robustness to the protocol when compared to the LCP. In this case, a Byzantine process has no room to "lie" since it has to justify all of its proposals with a vector of signatures received from the other processes. The best a malicious process can do is to force correct processes to verify more signatures by sending proposals with invalid justifications. A correct process has to verify an extra vector of signatures beyond the $n - f$ threshold for each invalid justification it

receives before gathering $n - f$ valid proposals. Nevertheless, this does not have a significant performance impact because the cost of verifying signatures is much smaller than the cost of constructing signatures.

Number of Rounds. Table 2 shows the average number of rounds the protocols need to execute in order to reach a decision and terminate. The results are the average number of rounds until decision of all consensus executions of the above experiments, which add up to approximately 6000 executions per each tested group size/faultload pair.

	Local Coin		
	$n = 4$	$n = 7$	$n = 10$
failure-free	1,004 (0,42)	1,005 (0,14)	1,009 (0,19)
fail-stop	1,000 (0)	1,000 (0)	1,000 (0)
Byzantine	1,462 (1,52)	1,569 (1,69)	2,289 (2,79)
	Shared Coin		
	$n = 4$	$n = 7$	$n = 10$
failure-free	1,013 (0,23)	1,018 (0,27)	1,010 (0,20)
fail-stop	1,000 (0)	1,000 (0)	1,000 (0)
Byzantine	1,016 (0,25)	1,017 (0,26)	1,012 (0,22)

Table 2. Average number of rounds for approximately 6000 executions. The standard deviation is shown in parenthesis.

The first noticeable result is that the average number of rounds is much lower than what is suggested by previous theoretical results even when considering Byzantine faults. For instance, the expected theoretical number of rounds for the LCP with a strong adversary is 2^{n-f}. The obtained average number of rounds with 10 processes (of which 3 of them are malicious) is a little bit above 2 rounds (2,289) which is very far from the theoretical result (128 rounds). A reason that can explain this difference between the observed and theoretical results is related to the kind of failures that are being considered. The theoretical result is obtained using the strong adversary model, where the adversary controls the scheduling of the network, which means that it can delay specific messages in order to postpone termination.

The performance of both protocols is similar except for the Byzantine case, in which the SCP is much more robust. In fact, there is practically no difference in the number of rounds of the SCP between the failure-free and Byzantine faultloads. The SCP also shows no degradation with an increasing number of processes, while the LCP shows some degradation, but only in the Byzantine scenario.

The fact that both protocols always reach decision in one round with f crashed processes has a simple explanation. When f processes crash it is guaranteed that at every step of the protocols, the correct processes always see the same set of expected $n - f$ messages. Since the proposed values at each step are based on this set of $n - f$ messages, they will always propose the same values every step of the way, thus achieving a decision by the first round.

5.4. Cryptography vs. Bandwidth

Although the previous results already give some idea of the impact of the cryptographic and bandwidth parameters on the performance of the protocols, this section provides a comprehensive analysis of the relative costs of communication and computation. In this experiment the only varying parameters are the bandwidth and the cryptographic settings. The tested bandwidth values are 1000, 100, and 10 Mbit/s. The measurements made for the LCP use IPSec activated and deactivated. For the SCP, the experiments are taken with three RSA key sizes: 512, 1024, and 2048 bits. No failures were injected during the experiments, the group size was 4 processes, and the proposal distribution was set to uniform.

Figure 7 depicts the performance of the LCP with each curve corresponding to a specific bandwidth/IPSec combination. It shows that cryptography (IPsec) has almost no impact on the performance of the protocol, and that reducing the available bandwidth results in a great performance cost. Every time the bandwidth is downgraded, the latency and throughput suffer a significant degradation. Considering 200 concurrent consensus executions, the throughput has a value of approximately 1300 decisions/s with a bandwidth of 1000 Mbit/s, 170 decisions/s with 100 Mbit/s, and 60 decisions/s for with 10 Mbit/s.

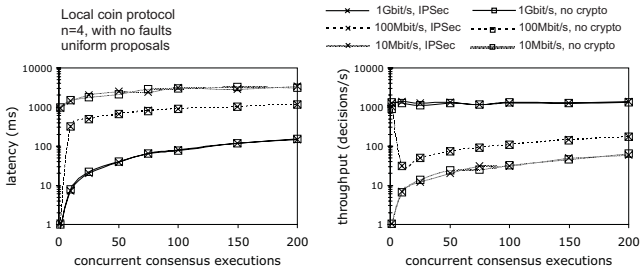

Figure 7. Burst latency and throughput for the LCP with different bandwidth and cryptographic parameters

The performance of the SCP is shown in Figure 8. This scenario is quite different from the LCP. While reducing the network bandwidth has a negative effect on performance, this cost is not as accentuated as the cost of increasing the key size. Regardless of the available bandwidth, the measurements with 2048-bit keys rank the bottommost among all the experiments, and the measurements with 512-bit keys rank among the first four spots. More closely, for 200 concurrent consensus executions, the measured throughput with a bandwidth of 1000 Mb/s was approximately 46, 30, and 10 decisions/s for 512, 1024, and 2048 bit keys, respectively. With 100 Mb/s it was 39, 23, and 8 decisions/s for

512, 1024, and 2048 bit keys, and with 10 Mb/s was 19, 16, and 6 decisions/s for 512, 1024, and 2048 bit keys.

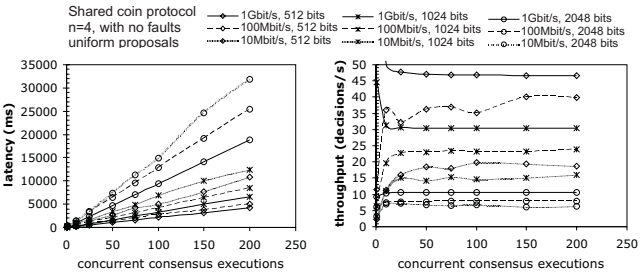

Figure 8. Burst latency and throughput for the SCP with different bandwidth and cryptographic parameters

The SCP, despite getting significantly closer to the LCP in terms of performance each time the network settings were downgraded, it never matched the performance of the LCP with identical network bandwidth. Nevertheless, in a WAN setting where there is much less bandwidth available, and there is a significantly higher network delay, it seems plausible for the SCP to outperform the LCP. Besides benefiting from its relatively lower number of exchanged messages, the network latency would offset the cryptographic computation costs of the SCP.

5.5. Summary of Results

The most important conclusions from the experimental evaluation are summarized in the following points:

- The LCP is significantly faster than the SCP with similar system parameters for all the environmental settings tested.
- The SCP, while slower, proved to be more scalable since its performance degraded to a lesser degree than the LCP with an increasing number of processes.
- The SCP is more robust than the LCP since it was not affected in terms of performance when malicious faults were injected in the system. The LCP evidenced a little degradation with respect to the number of rounds, latency and throughput.
- The measured average number of rounds of both protocols was quite small, being close to one with no faults, and exactly one with f crashed processes. With respect to the situation with f malicious processes, the SCP scored similar to the failure-free scenario, and the LCP showed a small degradation which was accentuated when the number of processes was higher.
- The performance bottleneck for the LCP, when many consensuses were executed concurrently, was the network because of its high number of exchanged messages and use of cheap cryptography. For the SCP, the bottleneck was the CPU because it exchanged a small number of messages and utilized expensive asymmetric cryptography. This makes the LCP a good contender for LAN settings, and the SCP a strong candidate for WAN environments, if the objective is to execute many consensuses simultaneously.

6. Related Work

The two classes of randomized (i.e., that employ random numbers) binary consensus protocols studied in this paper were introduced in 1983 in two seminal papers due to Ben-Or and Rabin [1, 17]. Ben-Or presented two algorithms, tolerating respectively crash and Byzantine (or arbitrary) faults. The Byzantine protocol had sub-optimal resilience (tolerated f faulty processes out of $n = 5f + 1$) and terminated in an expected exponential number of rounds. Rabin's protocol tolerated Byzantine faults and the resilience was also sub-optimal (f out of $n = 10f + 1$). In the years that followed, several randomized binary consensus protocols were presented, both following Ben-Or's local coin-style protocol [3] and Rabin's shared coin approach [18, 2, 6]. A detailed survey on this early work can be found in [8].

Research in randomized binary protocols has been mostly theoretical. Among the several goals pursued, we emphasize the quests for optimal resilience, constant expected round complexity, and better coin sharing schemes. Protocols with optimal resilience, i.e., that tolerate f out of $3f + 1$ faulty processes, were presented quite early, both for local coins [3] (one of the protocols considered in this paper) and for shared coins [18]. Constant expected round complexity was for a long time an objective of this line of research. The first protocol to attain this goal was presented by Canetti and Rabin [6]. This protocol is based on shared coins, but has a constant expected round complexity at the cost of an enormous number of transmitted messages (many thousands even for low numbers of processes). One of the protocols considered in this paper, ABBA, also terminates in a constant expected number of rounds but this number is low and the number of messages sent is much lower than Canetti and Rabin's [4]. In relation to coin sharing, the original Rabin's protocol had a need for a preliminary distribution of data among the processes for each coin tossed [17], something that can be quite impractical for real applications. This requirement was later removed. More recent protocols do not need this initial distribution of data [6, 4].

Although there are many randomized binary consensus protocols described in the literature, we are aware of a single implementation: the ABBA protocol, which has been

implemented in the context of the SINTRA system [5]. This implementation was done in Java. No measurements of the performance of ABBA were presented, only measurements of an atomic multicast protocol that executed many instantiations of ABBA. Moreover, these measurements were made only for the mean time between message receptions and did not include many system conditions like we have done here. Only failure-free executions were considered.

Recently the authors presented a study of the performance of a stack of protocols built on top of a binary randomized consensus [16]. The stack included primitives like multi-valued consensus, vector consensus and atomic multicast. The objective was to assess how such a stack would perform in practice with realistic faultloads. The study described in the present paper has a distinct purpose because it compares two different classes of randomized binary consensus protocols.

7. Conclusion and Future Work

Randomized consensus protocols can be divided in two classes, depending how the "coin tossing" operation is performed: using a local coin or a shared coin. Most work in this area has been theoretical and, in terms of theoretical metrics, shared coin protocols usually perform much better, i.e., they typically have a lower complexity. However, the fact that these protocols usually utilize asymmetric cryptography can have a considerable impact on the performance, especially in LANs, where the communication delay is low. Getting a better understanding about these tradeoffs was one of the main motivations for the present work.

Our experimental evaluation lead to several important conclusions. The first is that the local coin protocol has a much better latency and throughput than the shared coin protocol in a LAN, both with and without crash or Byzantine failures. Another conclusion is that the higher number of messages sent by the local coin protocol tends to increase the latency and decrease the throughput when many consensuses are executed in parallel, especially when the bandwidth is reduced. Shared coin protocols are somewhat less sensitive to these factors, since much of their cost is in computing cryptographic operations in the machines.

These final results give the idea that shared coin protocols potentially perform better than local coin protocols in a WAN, since the available bandwidth is usually much lower than the minimum we considered in the experiments (10Mbit/s). Moreover, since the communication delay is typically much higher, the extra rounds will have an important cost. As future work, we plan to make an experimental assessment of the protocols in a WAN environment.

References

[1] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*, 1983.

[2] M. Ben-Or. Fast asynchronous Byzantine agreement. In *Proceedings of the 4th ACM Symp. on Principles of Distributed Computing*, pages 149–151, Aug. 1985.

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

[4] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proc. of the 19th ACM Symp. on Principles of Distributed Computing*, 2000.

[5] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 167–176, June 2002.

[6] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. of the 25th Annual ACM Symp. on Theory of Computing*, pages 42–51, 1993.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 1996.

[8] B. Chor and C. Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, pages 443–497. JAI Press, 1989.

[9] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[10] M. Correia, N. F. Neves, and P. Verssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 2006.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[12] 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, Apr. 1985.

[13] J. S. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, Aug. 1985.

[14] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF Request for Comments: RFC 2093, Nov. 1998.

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

[16] H. Moniz, M. Correia, N. F. Neves, and P. Verssimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.

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

[18] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, Aug. 1984.

[19] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*. Springer-Verlag, 2003.