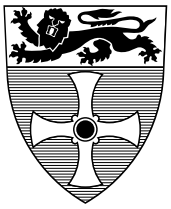
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

A Performance Study on the Signal-On-Fail Approach to Imposing Total Order in the Streets of Byzantium

Q. Inayat and P. Ezhilchelvan.

TECHNICAL REPORT SERIES

No. CS-TR-967 June, 2006

TECHNICAL REPORT SERIES

No. CS-TR-967 June, 2006

A Performance Study on the Signal-On-Fail Approach to Imposing Total Order in the Streets of Byzantium

Qurat-ul-Ain Inayat and Paul D. Ezhilchelvan

Abstract

Any asynchronous total-order protocol must somehow circumvent the well-known FLP impossibility result. This paper exposes the performance gains obtained when this impossibility is dealt with through the use of abstract processes built to have some special failure semantics. Specifically, we build processes with signal-on-fail semantics by (i) having a subset of Byzantine-prone processes paired to check each other's computational outputs, and (ii) assuming that paired processes do not fail simultaneously. By dynamically invoking the construction of signal-on-fail processes, coordinator-based total-order protocols which allow less than one-third of processes to fail in a Byzantine manner are developed. Using a LAN-based implementation, failure-free order latencies and fail-over latencies are measured; the former are shown to be smaller compared to the protocol of Castro and Liskov which is generally regarded to perform exceedingly well in the best-case scenarios.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

INAYAT, Q., EZHILCHELVAN, P..

A Performance Study on the Signal-On-Fail Approach to Imposing Total Order in the Streets of Byzantium [By] Q. Inayat and P. Ezhilchelvan

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-967)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-967

Abstract

Any asynchronous total-order protocol must somehow circumvent the well-known FLP impossibility result. This paper exposes the performance gains obtained when this impossibility is dealt with through the use of abstract processes built to have some special failure semantics. Specifically, we build processes with signal-on-fail semantics by (i) having a subset of Byzantine-prone processes paired to check each other's computational outputs, and (ii) assuming that paired processes do not fail simultaneously. By dynamically invoking the construction of signal-on-fail processes, coordinator-based total-order protocols which allow less than one-third of processes to fail in a Byzantine manner are developed. Using a LAN-based implementation, failure-free order latencies and fail-over latencies are measured; the former are shown to be smaller compared to the protocol of Castro and Liskov which is generally regarded to perform exceedingly well in the best-case scenarios.

About the author

Qurat-ul-Ain Inayat is a PhD student at the School of Computing Science, University of Newcastle upon Tyne. She obtained a BE (Computer Systems) and an M.Engg. (Computer Systems) degree from the Department of Computer and Information Systems Engineering, NED University of Engineering and Technology, Karachi, Pakistan. She is currently an assistant professor at NED University and is currently on study leave from there for her PhD. Her current area of research is Fault-tolerant replication management protocols. Her other areas of interest are distributed algorithms, fault and intrusion tolerance in distributed systems, performance evaluation and computer security.

Paul Devadoss Ezhilchelvan received Ph.D. degree in computer science in 1989 from the University of Newcastle upon Tyne, United Kingdom. He received the Bachelor of Engineering degree in 1981from the University of Madras, India, and the Master of Engineering degree in 1983 from the Indian Institute of Science, Bangalore. He joined the School of Computing Science of the University of Newcastle upon Tyne in 1983 where he is currently a lecturer.

Suggested keywords

MESSAGE ORDERING,
DISTRIBUTED CONSENSUS,
FAIL-SIGNAL,
IMPLEMENTATION
PERFORMANCE EVALUATION.

A Performance Study on the Signal-On-Fail Approach to Imposing Total Order in the Streets of Byzantium

Qurat-ul-Ain Inayat and Paul Devadoss Ezhilchelvan
School of Computing Science
University of Newcastle, Newcastle upon Tyne, UK
{q.inayat, paul.ezhilchelvan}@ncl.ac.uk

Abstract

Any asynchronous total-order protocol must somehow circumvent the well-known FLP impossibility result. This paper exposes the performance gains obtained when this impossibility is dealt with through the use of abstract processes built to have some special failure semantics. Specifically, we build processes with signal-on-fail semantics by (i) having a subset of Byzantine-prone processes paired to check each other's computational outputs, and (ii) assuming that paired processes do not fail simultaneously. By dynamically invoking the construction of signal-on-fail processes, coordinator-based total-order protocols which allow less than one-third of processes to fail in a Byzantine manner are developed. Using a LAN-based implementation, failure-free order latencies and fail-over latencies are measured; the former are shown to be smaller compared to the protocol of Castro and Liskov which is generally regarded to perform exceedingly well in the best-case scenarios.

1. Introduction

Managing service or state machine replication in the presence of faults requires that the non-faulty replicas be enabled to determine an *identical order* on client requests [18]. We address this ordering requirement when nodes hosting replicas can fail in a malicious, Byzantine manner and are connected by an *asynchronous* network, e.g., the Internet, wherein the message transfer delays cannot be bounded with certainty by a known constant. In particular, we propose, and evaluate the benefits of, a novel approach to circumvent the well-known FLP impossibility [5] which states that the ordering requirement cannot be met deterministically if the network is asynchronous and if replicas fail even merely by crashing, i.e. stopping to function in a quiescent manner.

Several order protocols appear in the literature (e.g., [21, 16, 12, 3]), circumventing the FLP impossibility in distinct ways. Of them, the Byzantine Fault-Tolerant order protocol by Castro and Liskov [2], denoted here as BFT, has been shown to have outstanding performance, particularly when there are no failures. BFT is a co-ordinator based deterministic protocol and requires a partially synchronous network [4] wherein message delays eventually stabilise to an estimated bound for a sufficiently long period of time. The protocols developed here are shown to perform faster than BFT and also with a smaller message overhead in failure-free scenarios. Furthermore, they allow most of the replicas to be connected by an asynchronous network.

Our approach to dealing with the FLP impossibility is to dynamically construct an abstract process with *signal-on-crash* semantics: it fails only by crash and additionally *fail-signal*s its own imminent crash. When failures are signalled, the impossibility result ceases to apply and when they do not involve producing incorrect outputs, a simplified protocol structure, smaller latencies and lower message overhead ensue. Construction of such a process however requires additional assumptions of a particular class not uncommon in the literature (e.g., [19, 6, 13, 7]): processes of the system are grouped and the members of a group are assumed to exhibit some prescribed failure behaviour. In our case, a subset of processes in the system is paired and the paired processes cannot fail at the same time. To this end, we consider two alternative assumptions: given that one process in a pair has failed, the other remains non-faulty (1) for some specified minimum amount of time (*1_after_1*) or (2) for ever (*never_2_Fail*).

The paired-up processes, on being called upon, construct an abstract, signal-on-crash process similar

to the way the component processes of the abstract fail-stop process [19] maintain the signal-on-crash property. They operate in parallel and endorse each other's outputs if the latter are found consistent with own outputs. The endorsed outputs are treated as the outputs of the signal-on-crash process and the endorsement is indicated through digital signatures that are assumed to be non-forgeable. If a process within a pair suspects a failure of its counter-part, it stops all activities related to implementing the signal-on-crash process abstraction and indicate this stopping by outputting a *fail-signal* message. Thus, the signal-on-crash process either outputs verifiably-endorsed messages of correct contents or stops functioning after signalling its stopping. That is, it can only crash and when it does, it *fail-signal*s prior to doing so.

A requirement for failure detection within a process pair is that each process be able to detect the absence of an output expected from the other. Otherwise, an indefinite waiting will result if the expected output is not produced. So, we make assumptions concerning the communication delays and the relative processing delays between processes of a given pair. We consider two delay models: the *synchronous model* (*Sync*) and the *partially synchronous* model [4] (*PSync*). We develop protocols for two combinations of assumptions, {*1_after_1*, *Sync*} and {*never_2_Fail*, *PSync*}, and study their performance.

The paper is structured as follows: the next section presents the system model and assumptions. Section 3 describes the signal-on-crash process construction and Section 4 the order protocol for {*1_after_1*, *Sync*} first and then for {*never_2_Fail*, *PSync*}. Section 5 measures and presents the protocols' fault-free performance in relation to both BFT and a simple crash-tolerant protocol, and the fail-over latencies. Section 6 concludes the paper by examining the literature for related work and identifying the limitations of our approach.

2. System Model

We suppose that a service, constructed as a deterministic state machine, is (as shown in Figure 1) replicated over $(2f+1)$, $f \geq 1$, nodes connected by a network that is reliable and also asynchronous (like the Internet) in transferring messages. That is, every sent message is delivered uncorrupted at destination(s) and message transfer delays are finite but cannot be guaranteed to be bounded by a known constant.

The *replica* nodes are assumed to fail independently of each other, and f_r, $0 \leq f_r \leq f$, of them can fail in a (cryptography-constrained) Byzantine manner. Despite this, the service is to be made fault-tolerant by meeting the well-known state machine replication requirements [18]; the hardest one amongst them is to ensure that non-faulty replicas process identical sequence of *requests* sent by the clients of this (replicated) service, and this will be the sole focus of this paper. For this reason, we regard that the clients are correct and the processing of client requests is assumed not to require availing of services provided elsewhere. (See [11] for making multi-tiered service provisioning Byzantine fault tolerant.)

Of the $(2f+1)$ replica nodes, at least f nodes are chosen to be supplemented with a *shadow* node solely for achieving fault-tolerant ordering of client requests over an asynchronous network. So, as in [22], these shadow nodes are a part of the Asynchrony-Byzantine (AB) order supplement scheme (See Fig 1) which is distinct from the Byzantine fault-tolerant service replication scheme (s_1, s_2, .. , $s_{(2f+1)}$).

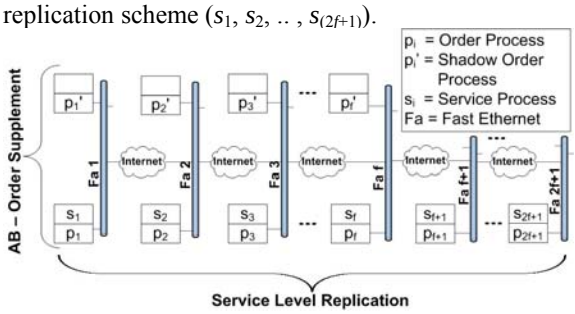


Figure 1. System Architecture.

We will denote a process that executes the order protocol in the i^{th} replica node as p_i, $1 \leq i \leq (2f+1)$, and adopt the convention of using $'$ to denote the order process in a shadow node, e.g., p'_i denotes the shadow of p_i. Note that there is no s'_i. The paired processes $\{p_i, p'_i\}$ may be called on to implement signal-on-crash process; to this end, their host nodes are connected to each other by a fast reliable network. The communication between any p'_i and p_j or p'_j, $i \neq j$ is over the reliable asynchronous network of the service level replication scheme. Thus, all forms of communication in our system except those between paired nodes conform to the classical asynchronous model. We make three major assumptions that are stated below. The first two are commonly made in the known Byzantine fault-tolerant solutions to message ordering in asynchronous networks.

2.1. Assumptions

Assumption 1. Shadow nodes fail independently and also in a Byzantine manner. The total number of failed shadow nodes does not exceed f_s, $0 \leq f_s \leq f$, such that

the total number of failed nodes in the system never exceeds f: $f_r+f_s \leq f$.

When exactly f service replica nodes are paired with a shadow, we have the optimum requirement for Byzantine fault-tolerant ordering: $n = 3f+1$, where n is the total number of nodes in the system.

Assumption 2. We assume that the known cryptographic techniques, such as public-key RSA signatures and message authentication codes [20], are robust enough to prevent message spoofing and replays and to detect message corruption. Specifically, a non-faulty process' signature for a given message cannot be forged and any attempt to alter a message signed by a non-faulty process will be detected. Moreover, the hash-functions used are assumed to be 1-way and collision-resistant: it is not possible to compute information from the digests produced using these functions or to find more than one message with the same digest. Finally, we assume that a *trusted dealer* initializes the system and the nodes with cryptographic keys and hash functions.

2.1.1. Assumptions specific to signal-on-crash approach. Say, order processes p and p' are paired to implement a signal-on-crash process. Essentially, each checks the other's output from both value and timing perspectives and endorses it if it is deemed correct relative to the locally generated output or emits a *fail-signal* otherwise. Constructing a signal-on-crash process in this way requires making assumptions about two timing-related aspects. The first aspect is about the ability to accurately estimate a differential delay bound within which one order process that has produced an output can expect its counter-part to do the same, if the counter-part is also operating in a timely manner and an output is expected as per the order protocol. Obviously, if an accurate estimation of this bound cannot be guaranteed, then correct p and p' can falsely suspect each other of untimely behavior and the abstract process will be falsely indicated to have 'crashed'. One could design protocols ruling out this false signaling altogether or by coping with it. The latter will involve making a relatively weaker assumption on the bound estimation process.

Signal-on-crash semantics cannot be realized if both p and p' can fail simultaneously. The second aspect is therefore about eliminating this possibility by defining a minimum time interval in which one node remains non-faulty while the other has failed. An assumption becomes very strong if this interval is taken to be infinity. Below, we make two distinct sets of assumptions; each set has a stronger assumption regarding one aspect and a weaker one on the other.

***Assumption 3(a)*:**

(i) The delay estimates used for assessing the timeliness of an order process are accurate and non-faulty processes never judge each other to be untimely. (*accurate delay estimation*)

(ii) The processes p and p' within any given node pair do not fail 'simultaneously': if one of them, say, p fails then p' does not fail at least until it observes the failure of p and an interval of $2D$ time elapses subsequent to the observation, where D is the unknown (but finite) bound on the communication delays over the reliable asynchronous network. (*sequential failure pattern*)

3(a)(i) means that when an order process of a non-faulty node does not receive an expected response from its counter-part within the delay estimate used, then the other node has become faulty at that moment.

***Assumption 3(b)*:**

(i) The delay estimates used for assessing the timeliness of an order process become accurate eventually: when the nodes within a pair remain non-faulty, there is an unknown timing instance after which neither order process will find the other untimely. (*eventually accurate delay estimation*)

(ii) At least one of the ordering processes p and p' does not fail. (*at most one fault*)

Remarks: *Assumptions on delay estimation accuracy*. 3(a)(i) regards the node-pair as a well-provisioned and well-engineered distributed system (e.g., [9]) and the delay estimates are *always* accurate. This view conforms to the traditional, albeit restrictive, *synchronous model*. 3(b)(i) allows estimated delays to become inaccurate occasionally together with a condition that the estimates eventually become accurate. This is the characteristic of the *timed asynchronous* [1] and *partially synchronous* [4] models.

Obviously, 3(b)(i) is weaker, and hence easier to realize, than 3(a)(i). A major implication of 3(b)(i) is that prior to the unknown timing instance, non-faulty order processes within a node pair may find each other untimely and consequently emit fail-signal to indicate the 'crash' of the *signal-on-crash* process they implement. If they later find each other timely, they can optimistically assume that the unknown timing instance has passed and resume implementing the *signal-on-crash* process. This amounts to the abstract process being restarted after a crash. Thus, the second assumption leads to *signal-on-crash* and *recovery* semantics, and further details can be seen in sub-section 4.4.

Remarks: *Assumptions on node failures*. Both 3(a)(ii) and 3(b)(ii) rule out both the nodes of a pair failing

simultaneously, say, due to the same underlying cause e.g., a failure of common power supply or a common design flaw that can be exploited by an attacker. They require exercising measures to ensure failure independence between the nodes and in particular eliminating any possible common failure modes through means such as diversity of node hardware and operating systems and housing the nodes at distinct locations. With fail-independence sufficiently assured, 3(a)(ii) is realistic in practice. For example, when nodes of a pair fail independently with an exponential failure rate μ, the probability that both nodes will not fail within $2D$ time of each other turns out be 99.999923% and 99.9998% when D is 1 and 2.5 seconds respectively with $1/\mu$ being 30 days. 3(b)(ii) however is a stronger assumption since it expects at least one node to remain non-faulty throughout the mission time. The larger is the latter, the less likely that it will hold. For longer operative periods, we require that the signal-on-crash process be built using more than two processes when 3(b)(ii) is assumed; more precisely, each of the selected replica nodes (see Figure 1) needs to be supplemented with ϕ, $\phi > 1$, shadow nodes and at most ϕ nodes can fail in a given AB-order supplement group.

3. The Signal-on-Crash Set-up

In this set-up, assumptions 3(a)(i) and 3(a)(ii) hold and exactly f replica nodes are paired with one shadow. Thus, the set of processes executing the order protocol is $\{p_1, p_2, .. , p_{(2f+1)}, p'_1, p'_2, .. , p'_f\}$ and there are $n = 3f+1$ processes in total. We reserve the term *doubly-signed* to mean that a message is signed by two processes in sequence: the second process considers the signature of the first as a part of the contents it signs for. Thus, through a double-signed message, the second signatory can indicate its approval on the contents of the message that the first signatory has computed. Finally, clients are assumed to direct their requests to all nodes and thus all non-faulty processes receive each request that needs to be sequenced before processing.

3.1. Mutual Checking and Output Endorsement

Each of the paired processes, e.g., p_i or p'_i, $1 \leq i \leq f$, executes the protocol like any unpaired process, p_i, $f+1 \leq i \leq 2f+1$, in addition to collaborating with its paired counter-part to implement a signal-on-crash process. This collaboration in normal form involves each process (i) forwarding to its counterpart process a copy of every message it receives and sends over the asynchronous network, and (ii) verifying if the messages sent by the other process are correct (as per the order protocol) in value domain, and also correct in the time domain (using the delay estimate).

The collaboration between paired processes takes a more active form when the pair acts as the coordinator of the order protocol. (Note that our protocol, like BFT [2], is coordinator-based and deterministic.) When the pair $\{p_i, p'_i\}$ acts as the co-ordinator, p_i decides an order for each unordered client request and forwards its signed decision only to its shadow p'_i. (See Fig. 2.) If the latter finds the order decision of p_i to be valid (i.e., observes no value-domain failure), then it endorses the decision by double-signing it and sending the doubly-signed decision to all processes (including p_i). When p_i receives an authentic, doubly-signed message from p'_i, it forwards the received to all other processes (including p'_i). p'_i will also monitor whether p_i is deciding an order for every request which it has forwarded; not deciding an order will constitute a time-domain failure by p_i. It is easy to see that the paired processes operate together as a single non-faulty coordinator (except for the doubly-signed output format), so long as no non-faulty process in the pair observes a failure on its counter-part.

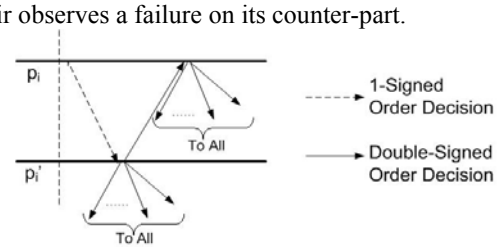


Figure 2. Order Endorsement.

3.2. Fail-Signalling on 'Crash'

At the time of system initialisation, each paired process is assumed to have been supplied with a *fail-signal* message signed by its counter-part. On detecting a value- or time-domain failure, a process double-signs the *fail-signal* it has been supplied with, and broadcasts the doubly-signed *fail-signal* to all other processes (including its counter-part). Similarly, when a process receives an authentic, doubly-signed *fail-signal* from its counter-part, it also double-signs the *fail-signal* it has and broadcasts the doubly-signed message. After having emitted *fail-signal*, processes stop their collaboration for implementing the signal-on-crash process.

An authentic, doubly-signed *fail-signal*, on being received, informs a destination process that the signatories of the received *fail-signal* decided not to work as a pair any longer (but will continue to operate

as individual entities). It also causes the destination process to echo the *fail-signal* to the first signatory in case the second signatory has (maliciously) omitted to send the *fail-signal* to its counterpart. Thus, a *signal-on-crash* process, implemented through mutual-checking of outputs, doubly-signed endorsement and fail-signaling, has the following properties:

SC1: Any authentic, doubly-signed message from a process pair is uniquely attributable to the source and contains correct information.

SC2: An authentic doubly-signed *fail-signal* is a definitive indicator of one faulty process in the source pair.

SC3: If a non-faulty process receives an authentic doubly-signed *fail-signal* from a given pair, every non-faulty is guaranteed to receive an authentic doubly-signed *fail-signal* from that pair.

4. The Order Protocol with Signal-on-Crash Set-up

The protocol defines a fixed set of f+1 sequentially-ranked co-ordinator candidates, $\{C_c: 1 \leq c \leq (f+1)\}$, comprising all f process-pairs and a randomly-chosen unpaired process. The former are ranked prior to the latter (the unpaired process). For the sake of exposition, let C_c be $\{p_c, p'_c\}$ for $1 \leq c \leq f$ and C_{f+1} be p_{f+1}.

Each process has a variable c that holds the rank of the candidate currently acting as the coordinator and is initialized to 1. That is, the protocol initially assigns C_1 as the coordinator. C_c, $2 \leq c \leq (f+1)$, can take over the coordinator role only if all $C_1, .. , C_{c-1}$ have fail-signalled, i.e., only if each pair $\{p_i, p'_i\}$, $1 \leq i \leq (c-1)$, has emitted an authentic, double-signed *fail-signal*. By SC2, a fail-signalling process-pair has a faulty process in it. Therefore, when the unpaired process p_{f+1} takes over as the $(f+1)^{th}$ coordinator, it must be non-faulty and remain so. So, the processes readily accept the ordering decisions of the $(f+1)^{th}$ coordinator. In what follows, we would describe the more challenging part of the protocol involving only the first f coordinators.

When $c = i$, the process pair $\{p_i, p'_i\}$ operates as the coordinator as described earlier: p_i decides a unique, in-sequence order for each client request which, if found valid, is endorsed by p'_i (see Figure 2); authentic, doubly-signed order decisions are thus transmitted by both p_i and p'_i to other processes. Producing doubly-signed order decisions constitutes the first two phases of our protocol and is shown in Figure 3(a) where the pair $\{p_1, p'_1\}$ is acting as the coordinator. Such an order decision will be denoted as order$<c, o, D(m)>$ where o is the unique sequence-number assigned to the request m and $D(m)$ is a digest of m. Since clients are correct and direct their requests to all nodes, the order for m does not contain m itself.

A process that receives a doubly-signed order, executes the *normal* part of the protocol in an attempt to *commit* m to o: the request m *irreversibly* gets assigned to the sequence number o indicated in the order message. The coordinating process pair may emit a *fail-signal* which calls for the installation of the next coordinator. This installation is carried out by an execution of the *install* part of the protocol. The two parts of the protocol are described below.

4.1 The Normal Part of the Protocol

Any process p_i that has received an authentic, doubly-signed, in-sequence order executes the following steps:

N1: Multicast a signed ack (that also contains the received order) to all processes (including itself);

N2: Wait until ack or order is received from at least (n-f) distinct processes;

N3: Commit order and retain the (n-f) distinct ack/order received as a proof of commitment;

The last (commit) phase of Figure 3(a) shows the execution of steps N2 and N3. Figure 3(b) depicts the three phases which the BFT will take to commit an order to facilitate an easy comparison. (Replica 1 is acting as the BFT coordinator.) Note that the three phases of BFT involve: 1 to n (coordinator to all), n to n, and again n to n message transmission. The purpose of the *prepare* phase is to verify if the coordinator can be trusted in what it sent during the *pre-prepare* phase. When a process-pair (with *signal-on-crash* semantics) is acting as the coordinator, n to n

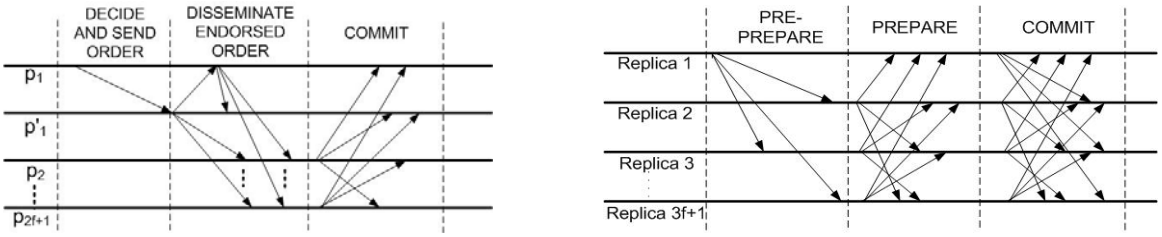


Figure 3. Fail-Free 3-phase Operation: (a) Our Protocol. (b) BFT.

transmissions of the *prepare* phase is obviously not needed and the 3-phased exchanges become: 1 to 1 (for endorsement), 2 to *n* (endorsed output to all), and *n* to *n*.

4.2. The Install Part of the Protocol

A process p_i that receives an authentic, doubly-signed *fail-signal* from p_c or p'_c executes these steps:

IN1: c:= c+1. Ignore any incoming order message until the new coordinator is installed. Prepare BackLog message containing (a) received *fail-signal*, (b) committed order with the largest sequence number (max_committed) together with the proof of commitment, and (c) all acked but uncommitted orders. Multicast BackLog to all processes.

IN2: If $p_i = p_c$ then wait until BackLog is received from at least (*n-f*) distinct processes. Compute NewBackLog and *start_o* as mentioned below and prepare a Start message containing those two. If p_c is a paired process, sign and send Start with all (*n-f*) BackLogs to p'_c else sign and multicast Start to all processes. p'_c verifies if p_c computed properly the Start as per the (*n-f*) BackLogs received with it. If Start is found valid, it is doubly-signed and multicast.

IN3: If $f > 1$, any p_i or p'_i, $i \neq c$, that receives an authentic doubly- signed Start, generates its signature for the received and sends its unique identifier and the signature to p_c and p'_c.

IN4: p_c and p'_c, after receiving identifier-signature tuples from (*f*-1) distinct processes (other than *themselves*), multicasts these tuples to all.

IN5: Any p_i that receives an authentic doubly-signed Start and (*f*-1) identifier-signature tuples, regards that the new coordinator has been installed, treats Start as an order message with sequence number *start_o* and executes the normal part of the protocol to get Start committed. Once committed, all order messages included in it are considered committed as well. Note if the max_committed of p_i is smaller than the smallest *o* of order messages contained in Start, it is possible that p_i has some order messages missing. In that case, p_i is guaranteed to receive each of those order messages from at least (*f*+1) correct processes due to the way NewBackLog is computed. So it waits for (*f*+1) agreeing order messages to be received.

The process p_c computes NewBackLog by first including the order that has the largest sequence number (*o*) amongst all the max_committed orders received in the (*n-f*) BackLogs. (Let max{max_committed} denote *o* of this order.) It then includes every uncommitted order present in any of the (*n-f*) BackLogs with sequence no. > max{max_committed}.

It is possible that p'_c finds order<*c*-1, *o, D(m)*> and order<*c*-1, *o, D(m')*>, $m \neq m'$, in the (*n-f*) BackLogs which it received from p_c (together with 1-signed Start). If so, it should verify whether p_c has chosen to put the 'right' order, if any, into the NewBackLog, where the right order is the one that might have been committed by some correct process. This verification is done using the BackLogs which p'_c received directly from other processes. Omitting the details for the sake of simplicity, we present only the principles underpinning this verification. If both order<*c*-1, *o, D(m)*> and order<*c*-1, *o, D(m')*>are doubly-signed and authentic, then both p_{c-1} or p'_{c-1} have failed and, by assumption 3(a)(ii), at least 2*D* time has elapsed subsequent to the first of these failures has been observed. So, if, say, order<*c*-1, *o, D(m)*>, is committed by some process, then p'_c will have at least (*f*+1) processes having included order<*c*-1, *o, D(m)*> in their backlogs and only at most *f* processes having included order<*c*-1, *o, D(m')*>.

4.3. Protocol Optimizations

This sub-section presents two optimisations. The first one seeks to reduce the number of processes injecting messages on to the network. By property SC3 and assumption 3(a)(i), a fail-signalled pair does have at least one failed process in it. So, every time a new coordinator is installed, the processes of the old coordinator are turned into 'dumb' processes which can execute the protocol but cannot transmit messages. The total number (*n*) of processes in the system is reduced by 2 to account for the new dumb processes and the maximum number (*f*) of faulty processes by 1.

The second optimisation aims to reduce the number of order messages and ack messages injected into the system through batching of order messages. The coordinator process p_c batches the doubly-signed order messages generated over a period of time, *batching-interval*, and transmit a batch of order messages; similarly, processes transmit ack messages for a batch of order messages. Note that *batch-size* can become large if a long *batching-interval* is chosen or if too many client requests are sent. The effects of varying *batching-interval* and *batch-size* are studied in Section 5.

4.4 Protocol Extension for Signal-on-Crash and Recovery Set-up

We suppose now that the assumptions of 3(b) hold instead of those of 3(a). Observe that only 3(b)(i) is weaker compared to 3(a)(i). So, the extension needs to

address only the implications that arise due to weakening of 3(a)(i), possibly by taking advantage of assumption 3(b)(ii): at least one process within a pair is always non-faulty.

A major implication of 3(b)(i) being weaker is that paired-processes p_c and p'_c may find each other untimely even if both are non-faulty. So, after having fail-signalled, if they subsequently find each other timely through their continued mutual-checking (see Section 3.1), they should work as a pair if the need arises. Thus, SC2 holds no longer. Each process p_c or p'_c maintains a status variable $status_c$ that indicates the operative status of the pair: {*up*, *down*, *permanently_down*}. $status_c$ is irreversibly set to *permanently_down* when a process observes a value domain failure of its counter-part.

Since SC2 does not hold, it can no longer be ascertained that f failures have occurred if f distinct process-pairs have fail-signalled, and therefore when the un-paired p_{f+1} becomes the coordinator it cannot be expected to be non-faulty. So, (at least) (f+1) process-pairs are now required and we assume that p_{f+1} is paired with p'_{f+1}, bringing $n = 3f+2$; furthermore, only paired processes are allowed to act as coordinators. Note that there will be at least one process pair in which both processes are non-faulty and see each other timely starting from some unknown time. So, eventually, there is a process-pair whose operative status will be always *up*. However, until that *always-up* pair emerges and becomes the coordinator, the system can be in an unstable state, calling for frequent coordinator changes. We propose to use the view-change part of BFT protocol, except for the following modifications:

For view v, the pair $\{p_c, p'_c\}$ is the coordinator candidate where $c = (v \bmod (f+1))$ if $(v \bmod (f+1)) \neq 0$, $c = (f+1)$ otherwise. If p_c or p'_c does not have $status_c = up$ when a ViewChange(v) message is received and therefore does not want to act as the coordinator for the proposed new view v, then it multicasts an Unwilling(v) message which includes the *fail-signal* message as well. Any process that receives Unwilling(v) echoes it back to both p_c or p'_c and, as in the BFT protocol, multicasts a ViewChange(v+1) message. (Note that non-coordinator processes do not wait on timeout: they either expect view v to be installed or Unwilling(v) to be received.) If, on the other hand, p_c and p'_c have $status_c = up$, p_c acts as the 'primary process' of the BFT protocol, with all its messages get endorsed and multicast by p'_c as shown in Figure 2.

5. Protocol Implementation and Performance Study

The protocols were implemented in Java using JDK 1.5 and on a cluster of 15 Linux machines (Fedora Core 4) connected by a LAN. Each machine has a 2.80 GHz Pentium IV processor and 2GB RAM. Paired processes communicate using RMI and the unpaired ones using TCP/IP sockets. The performance study has two parts: the first is of comparative nature in the best-case scenario and the second involves assessing our protocols' ability to deal with failures.

The comparative performance study considers our protocols, BFT and a crash-tolerant protocol, denoted as CT. The best-case scenario for all these protocols is: no failures and also no suspicions of failures (see also [2]). In this scenario, the unknown timing instance of assumption 3(b)(i) (in sub-section 2.1.1) is the system start-up time itself, i.e., 3(b)(i) becomes the same as 3(a)(i). So, the protocol developed for the Signal-on-Crash set-up (denoted from now on as SC) behaves identically to its extension for the Signal-on-Crash and Recovery set-up. Their distinction is meaningful only in the second part of the study. CT is simply derived from SC, with no process being paired and no cryptographic techniques used. Specifically, the shadow processes are excluded from the system (hence $n = 2f+1$), the coordinator process directly sends its order message to all other processes, and an order message is committed in the same way as SC. CT performance can therefore be used to see the extent of slow-down in BFT and SC when the type of faults tolerated switches from crash to Byzantine. The parameters we measure are precisely defined below.

Latency is the time interval between the instance the request is batched by the coordinator and the instance the first process commits a sequence number (o) for that request. This does not include the time duration a received request spends waiting to be batched.

Throughput is the number of messages committed by an order process per second.

Fail-over latency is measured as the time interval between the moment the current coordinator issues fail-signal and the instance the new coordinator issues a Start message with (f+1) identifier-signature tuples.

The parameters we vary are described below.

Batching interval is varied from 40 milliseconds (ms) to 500 ms, and the *batch_size* is fixed at 1 KB. **Cryptographic techniques**, used for these experiments, constitute of three distinct combinations of message digest and signature schemes: MD5 for taking message digests together with RSA scheme for

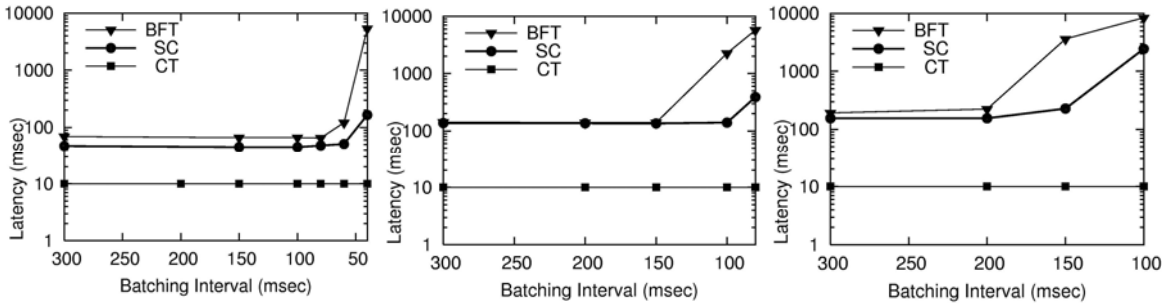


Figure 4: Order latency for f = 2 using (a) MD5 with RSA key size 1024 (b) MD5 with RSA key size 1536 (c) SHA1 with DSA key size 1024.

key sizes of 1024 and 1536, and SHA1 with DSA for the key size of 1024.

Fault- tolerance parameter (*f*) takes the value of 2 and 3.

BackLog size is varied between 1 KB to 5 KB to examine its effect on fail-over latency.

The results of our performance study are presented in the following manner. Figure 4 depicts order latency vs. batching interval for all three protocols, each for all three cryptographic techniques and *f* fixed at 2. Figure 5 shows throughput vs. batching interval for all three protocols with all three cryptographic techniques and *f* fixed at 2. Finally, figure 6 shows the fail-over latencies of the SC protocol and its extension for the Signal-on-Crash and Recovery set-up, which we denote as SCR. They are measured for various BackLog sizes, for all three crypto-techniques and for $f = 2$. Each point in a graph is an average over 100 experimental results.

Order Latency: As depicted in Fig 3, BFT has three phases to order a request involving: 1 to *n* (coordinator to all), *n* to *n*, and again *n* to *n* message transmissions; the SC has its three phases as: 1 to 1 (for endorsement within the pair), 2 to *n* (endorsed output to all), and *n* to *n* transmissions. Note that CT will only have the first and the second phases of SC combined into one (crash-prone) coordinator disseminating to all (1 to *n*) followed by *n* to *n* transmissions; also no cryptographic overhead is incurred.

Referring to figure 4, the order latencies stay nearly constant for large values of the batching intervals, indicating that the system operates in steady-state (i.e., in light or normal load conditions). They stay constant at 10 ms for CT, but increase drastically for BFT and SC when the batching interval decreases below a *threshold*, pushing the system operation into a 'saturation' region. (Note that latencies are represented in log scale along y-axis.) Further, the threshold for BFT is larger than that for SC. This indicates that BFT has a tendency to push the system into saturation earlier due to the large number of messages it places in the system and the cryptographic operations performed on each message. For the same reason, the steady-state latency for BFT is always more than that for SC. However, it is interesting to see that the differences in steady-state latencies of SC and BFT increase considerably when crypto-technique is changed from RSA (Figures 4a, 4b) to DSA scheme (Figure 4c). For example, RSA with key size 1024 gives a difference of 21ms between steady-state latencies of SC and BFT while the difference is 37ms when DSA scheme is used for authentication. The explanation is as follows. In both the schemes the time taken to sign a given message is similar; however, signature verification is much faster in the RSA scheme compared to DSA. Furthermore, in a typical *n* to *n* message exchange, each process signs one message while it needs to verify at least (*n*-*f*) messages. Therefore, there is a more pronounced slowing down of BFT due to the slowness of DSA verification. This suggests that DSA is generally not suited for Byzantine order protocols.

As we increase *f* to 3 (not shown here to avoid repetition), we observe similar trends, except that the saturation thresholds are encountered at larger *batching_intervals*, and the order latencies in the steady state increase. These observations can be attributed to the fact that as *n* increases, each individual process receives more messages which need to be authenticated and processed.

Throughput: Throughput was observed to be low for larger batching intervals for all the three crypto-techniques with *f*=2. Figure 5 shows that with decreasing batching intervals, throughput increases until the system reaches the saturation point after which it starts dropping down. This behavior was observed for both SC and BFT whereas the drop could not be observed for CT for the range of batching intervals used.

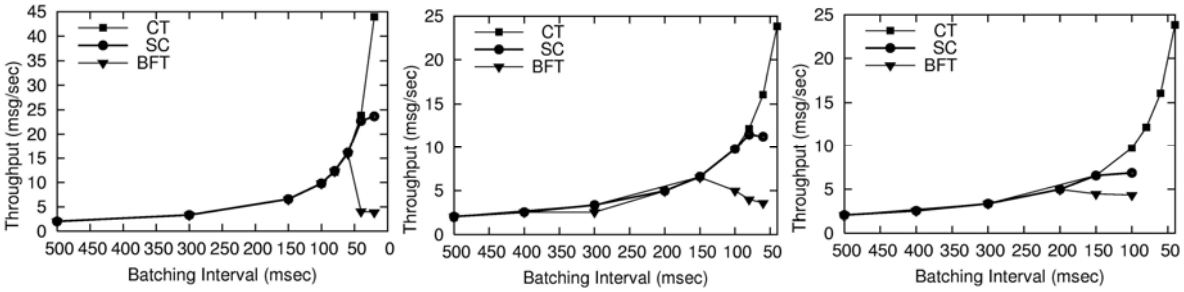


Figure 5: Throughput for f = 2 using (a) MD5 with RSA key size 1024 (b) MD5 with RSA key size 1536 (c) SHA1 with DSA key size 1024.

Observing the behavior of the three protocols for any of the crypto-technique, we confirm the conclusion drawn about BFT above that it causes system saturation earlier than the other two and throughput starts dropping immediately after entering saturation point which is found to stay stable for a while in case of SC. Here also saturation was observed to occur earlier with more expensive crypto-techniques.

Fail-Over Latency: Experiments were run to assess the effect of occurrences of faults on the performance of SC and SCR. A single value-domain fault was injected in the system and the duration for switching between the coordinators was measured for all three cryptographic schemes. It can be observed that the fail-over latency increases linearly with backlog size which was varied from 1 to 5 KB.

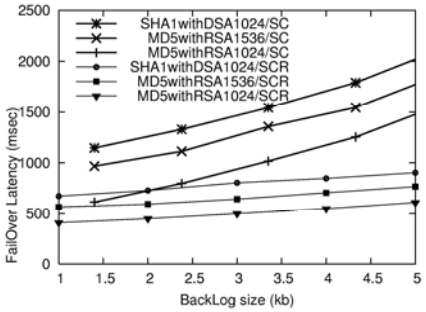


Figure 6: Fail-Over Latency in SC and SCR for f = 2 using all three crypto techniques

6. Related work and Conclusions

The FLP impossibility is typically circumvented by making synchrony assumptions (e.g., Rampart [15], SecureRing [8], TTCB [21], ITUA [16]), through quorum systems [12] or randomisation [3]. We note that some of the systems in the first category, like our protocol with assumption 3(a)(i), can violate safety if strong synchrony assumptions they make are not met, where safety is the property that non-faulty processes do not order requests differently. On the other hand, the BFT [2] and our extended protocol with assumption 3(b)(i), require weak synchrony assumptions only to guarantee liveness. Randomised protocols make no synchrony assumptions and liveness is guaranteed in probabilistic terms to be a certainty with the passage of time. Our protocols also make no synchrony assumptions between unpaired processes but offer deterministic liveness guarantees. (Assumptions 3(a)(i) and 3(b)(i) are concerned only with the paired processes.)

On the optimistic order protocols, some of the earlier work was done by Pedone and Schiper [14], which exploits not just the absence of failures but also the possibility of multicasts over a LAN being naturally received in the same order (spontaneous total-order). Following BFT [2], several optimistic Byzantine fault-tolerant order protocols were published (e.g., [23, 10, 17]), but none of them has been experimentally evaluated to the best of our knowledge. These protocols have the following design flavour: they are coordinator based in the normal part and when optimistic conditions are deemed not to hold, a (randomized) consensus protocol is executed to remedy the situation.

We have here developed an optimistic Byzantine fault-Tolerant order protocol that is demonstrated to perform better than a protocol best-known for its fail-free performance and practicability. We have achieved this by carefully applying a technique long-known for building robust process abstractions [19] that are easier to program with; abstract processes with signal-on-crash property, are deployed to act as coordinators for order protocols. Consequently, not only the order latency and the message overhead fall but also the protocol becomes easier to implement and no synchrony assumptions need to be made among the (un-paired) processes that do not have to cooperate to build the signal-on-crash abstraction. These benefits come at a cost: paired-up processes cannot fail simultaneously. We meet this requirement by assuming that if both processes fail, the failure

occurrences are separated by a threshold interval (assumption 3(a)(ii)) or that at least one process never fails (assumption 3(b)(ii)). We have argued that both the assumptions require implementation of measures that assure failure-independence, that 3(a)(ii) is realistic, and that a robust realisation of 3(b)(ii) involves using more than two processes to build the signal-on-crash abstraction. The latter in turn calls for making a trade-off between the (falling) hardware cost and the (much desired) performance benefits.

Acknowledgements: We thank Paul Murray for his meticulous shepherding of this paper. Financial support from the EPSRC platform grant is acknowledged. Qurat-ul-Ain is supported by Commonwealth Scholarship Commission for her PhD.

7. References

[1] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model", In IEEE Transactions on Parallel and Distributed Systems, Vol. 10 (6), June 1999, pp. 642-57.

[2] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", In Proceedings of the 3rd ACM Symposium on Operating Systems Design and Implementation (OSDI), February 1999, pp. 173-186.

[3] C. Cachin, K. Kursawe and V. Shoup, "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography", Proceedings of 19th ACM Symposium on Principles of Distributed Computing, 2000, pp. 123-132.

[4] C. Dwork , N. Lynch , L. Stockmeyer, "Consensus in the presence of partial synchrony", Journal of the ACM (JACM), v.35 n.2, p.288-323, April 1988.

[5] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," Journal of the ACM, Vol. 32, No. 2, pp. 374-382, April 1985.

[6] S. Haddad and F. Nguilla, "Combining Different Failure Detectors for Solving a Large-Scale Consensus Problem" In the proceedings of the 14th ISCA-CATA Cancun, Mexico, April 1999.

[7] F. Junqueira and K. Marzullo, "The virtue of dependent failures in multi-site systems", In Proceedings of the IEEE Workshop on Hot Topics in System Dependability, Supplemental volume of DSN'05, pages 242–247, June 2005.

[8] K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication", In Proc. of the 31st Annual Hawaii International Conference on System Sciences (HICSS), pp. 317-26, Jan. 1998.

[9] Kopetz H. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 1997, ISBN 0-7923-9894-7.

[10] K Kursawe and V Shoup, "Optimistic Asynchronous Atomic Broadcast", in the Proceedings of International Colloqium on Automata, Languages and Programming (ICALP05) (L Caires, G.F. Italiano, L. Monteiro, Eds.) LNCS 3580, pp. 204-215, Springer, 2005. (http://www.shoup.net/papers/ks.pdf).

[11] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, P. Narasimhan, "Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications", IEEE Symposium on Reliable Distributed Systems, Orlando, FL, 2005.

[12] D. Malkhi and M. Reiter, "Byzantine Quorum Systems", Distributed Computing, 11(4), pp.203-213, 1998.

[13] D. Mpoeleng, P.D. Ezhilchelvan and N.A. Speirs, "From Crash-tolerance to Authenticated Byzantine Tolerance: a Structured Approach, the costs and Benefits", Proc. 2003 International Conference on Dependable Systems and Networks (DSN2003), June 2003, pp.227-236.

[14] F. Pedone and A Schiper, "Optimistic Atomic Broadcast: A Pragmatic View-point" in Theoretical Computer Science (Elsevier), Vol. 291 (1), pp. 79-101, 2003.

[15] M. Reiter, "The Rampart Toolkit for Building High-Integrity Services". In Theory and practice of in Distributed Systems (LNCS 938), pp. 99 -110, Springer-Verlag, 1995.

[16] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems" Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002), Washington, DC, June 23-26, 2002, pp. 229-238.

[17] H V. Ramasamy and Christian Cachin, "Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast", In Proc. OPODIS 2005, 9th Intl. Conference on Principles of Distributed Systems, December 2005.

[18] F. Schneider and S. Toueg, "Replication Management Using the State-Machine Approach", in *Distributed Systems*, Second Edition, (Ed. S Mullender), Addison-Wesley, ISBN 0-201-62427-3, pp. 169-198.

[19] R. Schlichting and F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", ACM Transactions on Computer Systems, Vol. 1(3), pp. 222-238, August 1983.

[20] G Tsudik, "Message Authentication Using one-way Hash Functions", ACM Computer Communications Review, 22(5), 1992.

[21] P. Verissimo and A. Casimiro, "The Timely Computing Base Model and Architecture", IEEE Transaction on Computing Systems, 51(8), pp. 916-930, 2002.

[22] J.Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M Dahlin, "Seperating Agreement from Execution for Byzantine Tolerant Services", In proceedings of SOSP, pp. 253-267, 2003.

[23] K. Kursawe, "Optimistic Byzantine Agreement", In Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS 2002), October 2002, pp 262-267.