An Indulgent Uniform Total Order Algorithm with Optimistic Delivery

Pedro Vicente
Luís Rodrigues

DI–FCUL TR–02–13

17th September 2002

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

An Indulgent Uniform Total Order Algorithm with Optimistic Delivery[‡]

Pedro Vicente
Universidade de Lisboa
pedrofrv@di.fc.ul.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

17th September 2002

Abstract

A total order algorithm is a fundamental building block in the construction of distributed fault-tolerant applications. Unfortunately, the implementation of such a primitive can be expensive both in terms of communication steps and of number of messages exchanged. This problem is exacerbated in large-scale systems, where the performance of the algorithm may be limited by the presence of high-latency links. Typically, the most efficient total order algorithms do not provide uniform delivery and assume the availability of a perfect failure detector. Such algorithms may provide inconsistent results if the system assumptions do not hold. On the other hand, algorithms that assume an unreliable failure detector always provide consistent results but exhibit higher costs. This paper presents a new algorithm that combines the advantages of both approaches. On good periods, when the system is stable and processes are not suspected, the algorithm operates as if a perfect failure detector is assumed. Yet, the algorithm is indulgent, since it never violates consistency, even in runs where processes are suspected.

1 Introduction

A Total Order Broadcast algorithm is a fundamental building block in the construction of distributed fault-tolerant applications [1]. The purpose of such an algorithm is to provide a communication primitive that allows processes to agree on the set of messages they deliver and also on their delivery order. Total Order Broadcast is particularly useful to implement fault-tolerant services by using software-based replication [15]. By employing this primitive to disseminate updates, all correct copies of a service deliver the same set of updates in the same order, and consequently the state of the service is

*This work has been partially supported by the project IST-1999-20997, GlobData and by project POSI/CHS/41285/2001, StrongRep.

†Sections of this report have been published in the Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02). October, 2002, Osaka, Japan.

kept consistent. In particular, the algorithm presented in this paper is being used to support the replication of persistent object-oriented repositories on geographically dispersed nodes [27].

The Total Order Broadcast problem has been extensively studied in the literature. Therefore, several algorithms have been proposed to implement this primitive, considering a diverse set of execution environments and system models. Although many classifications are possible, when considering algorithms designed for asynchronous systems it is important two distinguish two main classes of algorithms: algorithms that assume a perfect failure detector and algorithms that assume only unreliable failure detectors [5]. In the first category, one must also distinguish algorithms that offer uniform total order from those that offer only non-uniform total order (the precise distinction between these two variants is defined in Section 3). Algorithms that assume unreliable failure detectors offer uniform total order.

Not surprisingly, the most efficient algorithms are the ones that assume a perfect failure detector and, among these, those that provide only non-uniform delivery [17, 2]. In this setting, is is possible to provide total order delivery in a single communication step: a sequencer based-algorithm [17] can deliver the messages from the sequencer in a single step, and a symmetric algorithm [20, 7, 2, 9] can also deliver messages in a single step when all processes are transmitting. Under the same assumptions, uniform delivery can be provided at the cost of one additional communication step, for a total of two communication steps. Unfortunately, if the assumptions do not hold, these algorithms may provide inconsistent results (or force processes to crash [4]).

On the other hand, algorithms that assume unreliable failure detectors are *indulgent* [14], *i.e.*, they never violate safety even if the assumptions do not hold (in the worst case, they may not terminate). It has been shown that, in this setting, the Total Order Broadcast problem is equivalent to the Consensus problem [5]. It has also been shown that Consensus cannot be solved in this type of systems in less than two communication steps [18]. Since many Total Order Broadcast algorithms for asynchronous systems with unreliable failure detectors use consensus as a building block, they require at least three communication steps [5, 26, 13, 28].

This paper presents a novel algorithm that combines the good properties of the algorithms in the two previous classes. In good periods, when the system is stable and processes are not suspected, the algorithm operates as an algorithm for the perfect failure detector model. Therefore, in stable periods (the most common case), the

algorithm is as efficient as the algorithms in the first class. When processes are suspected, the algorithm runs a consensus-based reconfiguration phase that ensures safety. Therefore, the algorithm is indulgent as it copes with partial synchrony and solves the Uniform Total Order problem assuming only unreliable failure detectors.

Additionally, the structure of the algorithm is exploited to provide *optimistic delivery*. An optimistic delivery is an early indication of the estimated uniform total order. The application can use this estimate to perform a number of actions optimistically, which are later committed when the final definitive order is established. The goal is to execute some application steps in parallel with the communication steps of the total order algorithm. Although several algorithms that support optimistic delivery have been proposed before [23, 10], these are specialized to some specific network types [23] or interaction patterns [10]. Our approach is more generic because, in stable conditions, both the optimistic and uniform order are derived from the output of a fully fledged non-uniform total order algorithm.

The principles of our approach are the following. An efficient algorithm that provides non-uniform total order assuming a perfect failure detector is used to provide fast optimistic delivery. If processes are not suspected, this optimistic order is made uniform through an additional round of message exchange. If processes are suspected, a consensus-based reconfiguration phase is executed to ensure the termination of pending broadcasts (determining a certain delivery order) and to reconfigure the operation mode for the next stable period. The reconfiguration procedure does not assume a perfect failure detector. The challenge of this algorithm is to ensure that the order established by the reconfiguration phase never conflicts with the order established during the stable-period.

The rest of the paper is structured as follows. Section 2 describes our system model and Section 3 defines the servide provided and introduces the building blocks used by the algorithm. A discussion on the cost of total order is provided in Section 4; it motivates our new total order algorithm which is presented in Section 5. Section 6 offers a performance evaluation of the algorithm, including measurements from a running prototype. Section 7 compares our work with related work and Section 8 concludes the paper.

2 System Model

2.1 Asynchronous System

We consider a distributed system composed of a finite set of processes $\Omega = \{p_1, p_2, \ldots, p_n\}$ completely connected through a set of channels. Communication is by message passing, asynchronous and eventually reliable. Asynchrony means that there is no bound on communication delays, nor on process relative speeds. An eventual reliable channel ensures that a message sent by a process p_i to a process p_j is eventually received by p_j, if p_i and p_j are correct (i.e., do not fail)[1]. An eventual reliable channel can be implemented by retransmitting lost or corrupt messages. Processes fail by crashing (we do not consider Byzantine failures). A correct process is a process that does not crash in an infinite run. We assume that only a minority of processes may crash, *i.e.*, let f be the maximum number of processes that can crash, then $f < |\Omega|/2$

2.2 Failure Detectors

Given the impossibility of reaching consensus in asynchronous systems [12], alternative system models have been defined: partially synchronous [8], timed asynchronous [11], and asynchronous augmented with failure detectors [5]. In this paper we follow the latter model. We consider failure detectors in class $\Diamond\mathcal{S}$ (*Eventually Strong*), the *weakest* class of failure detectors that allow to solve consensus and atomic broadcast/multicast problems [5]: such failure detectors can make an infinite number of false suspicions. Failure detectors are required to solve consensus and used to trigger the reconfiguration in our algorithm.

2.3 Local Clocks for Optimal Performance

Our algorithm does not require the use of physical clocks to ensure correctness. However, it adapts the configuration according to an estimate of the sending rate at each process and of the network delay among processes. In order to obtain these estimates, an implementation requires the use of local clocks with a stable drift rate. These clocks are used to measure the inter-arrival time of messages and to measure round trip delays. The inaccuracy of these clocks may result in sub-optimal configurations in terms of the average latency of message delivery but has no impact on the properties of the service.

[1] This does not exclude link failures, if we require that any link failure is eventually repaired.

UTO1 - Uniform Agreement: Consider *UTO-broadcast*(m). If a process in Ω (correct or not) has *UTO-delivered*(m), then every correct process in Ω eventually *UTO-delivers*(m).

UTO2 - Termination: If a correct process *UTO-broadcasts*(m), then every correct process in Ω eventually *UTO-delivers*(m).

UTO3 - Uniform Total order: Let m_1 and m_2 be two messages that are *UTO-broadcast*. We note $m_1 < m_2$ if and only if a process (correct or not) *UTO-delivers* m_1 before m_2. Total order ensures that the relation $<$ is acyclic.

UTO4 - Integrity: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by some process $p \in \Omega$.

Table 1: Uniform total order properties

3 Service Description and Building Blocks

In this section, we define the service provided by our algorithm and introduce some of the building blocks used in its design.

3.1 Uniform Total Order Broadcast with Optimistic Delivery

Uniform total order broadcast is defined on the set Ω by the primitives (1) *UTO-broadcast*(m) which issues message m to Ω, and (2) *UTO-deliver(m)* which is the corresponding delivery of m. When a process p_i executes *UTO-broadcast*(m) (resp *UTO-deliver(m)*), we say that p_i *"UTO-broadcasts m"* (resp *"UTO-delivers m"*). The properties of the primitive *UTO-broadcast* [16] are listed in Table 1.

Our algorithm includes an additional primitive *UTO-opt-deliver(m)*. When a process p_i executes *UTO-opt-deliver(m)*, we say that p_i *"UTO-opt-delivers m"*. The order by which a process p UTO-opt-delivers messages is an estimate of the order by which p will *UTO-deliver* the same messages. Note that in some cases the estimate may be wrong, *i.e.*, the order by which messages are *UTO-opt-delivered* may differ from the order by which they are *UTO-delivered* (although in stable periods it is desirable that it is the same). Note also that it is possible that a message is directly *UTO-delivered* without ever being UTO-opt-delivered.

3.2 Building Blocks

The building blocks of our algorithm are a Regular Reliable Broadcast primitive, a Consensus primitive and a Total Order algorithm

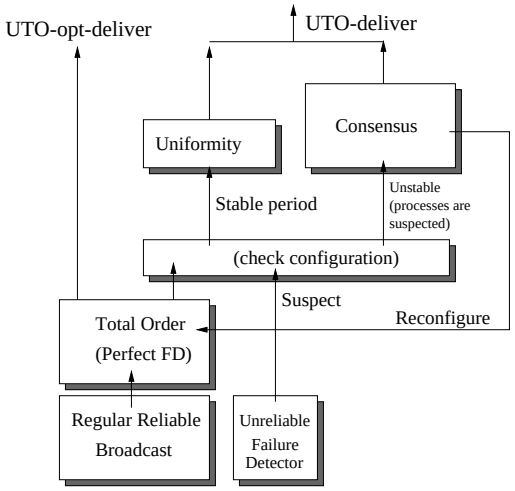

Figure 1: Algorithm Architecture.

primitive (for the Perfect Failure Detector model), as illustrated in Figure 1 (uniformity is achieved by running an additional round of message exchange).

Regular Reliable Broadcast We assume the existence of a (regular) *reliable* broadcast primitive, denoted *R-broadcast(m)*, and the corresponding reliable delivery primitive, denoted *R-deliver(m)*. The primitive *R-broadcast(m)* satisfies the following three properties: (1) (agreement) if a correct process in Ω has *R-delivered*(m), then every correct process in Ω eventually *R-delivers*(m); (2) (validity) if a correct process *R-broadcasts(m)*, then every correct process in Ω eventually *R-delivers*(m); (3) (integrity) for any message m, every correct process delivers m only if m was previously broadcast by some process $p \in \Omega$. An example of a simple implementation of reliable broadcast is given in [5].

Consensus We also assume the existence of a *consensus* function, which solves the uniform consensus problem [5]. The problem is defined over a set of processes Π, each proposes an initial value, and has to decide on a final value, such that (1) agreement: no two processes decide differently, (2) termination: every correct process eventually decides, and (3) non-triviality: the value decided is one of the values proposed. An example of a consensus algorithm for asynchronous systems augmented with unreliable failure detectors is given in [5].

Total Order with Perfect Failure Detector A fundamental goal of our design is to create an algorithm that, in stable periods, is as efficient as the algorithms that assume a perfect failure detector. Therefore, we have decided to select an existing algorithm in that class to serve as the basis for our new algorithm.

Although several algorithms have been described in the literature [4, 6, 7, 17, 19, 2], few were specifically targeted to operate in (geographically) large-scale systems. In a large scale network processes' traffic patterns are usually heterogeneous. The same applies to the network links: some processes will be located within the same local area network whereas others will be connected through slow links, and thus subject to long delays. In such an environment, none of the previous approaches can provide optimal performance. Therefore, we have opted to base our new algorithm on the Hybrid Total Order algorithm described in [25]. This algorithm combines two of the most used approaches to enforce total order in systems where a perfect failure detector is available, specifically: the *token-site* [6, 17] and *symmetric* [24, 7] approach. It does so by implementing a dynamic configuration policy that adapts the algorithm behavior such that the most adequate mechanism is used as a function of the network delays and of the traffic load.

In the token-based approach, a process is responsible for ordering messages on behalf of the other processes in the system. This process works as a sequencer of all messages and is often called the *token* holder. Token algorithms are appealing because they are relatively simple and provide good performance when message transit delays are small (they are particularly well suited for local area networks). However, in a token algorithm, a message sent by a process that does not hold the token experiences a latency of twice the network delay (i.e., the time to disseminate the message plus the time to obtain either the token or an order number from the token holder). Thus, token-based approaches are inefficient in face of large network delays. In the symmetric approach, ordering is established by all processes in a decentralized way, using information about message stability. This approach usually relies on *logical clocks* [20] or *vector clocks* [4, 24, 19]: messages are delivered according to their partial order and concurrent messages are totally ordered using some deterministic algorithm. Symmetric algorithms have the potential for providing low latency in message delivery when all processes are producing messages. In fact, using a technique called rate-synchronization [25], symmetric algorithms can exhibit a latency close to the network delay plus δ, where δ is the largest inter-message transmission time. Unfortunately, this also

RTO1 - Agreement: Consider *RTO-broadcast*(m). If a correct process in Ω has *RTO-delivered*(m), then every correct process in Ω eventually *RTO-delivers*(m).
RTO2 - Termination: (as UTO2).
RTO3 - Total order: Let m_1 and m_2 be two messages that are *RTO-broadcast*. We note $m_1 < m_2$ if and only if a correct process *RTO-delivers* m_1 before m_2. Total order ensures that the relation $<$ is acyclic.
RTO4 - Integrity: (as UTO4).

Table 2: Regular total order properties

means that all (or at least a majority [7, 2]) of processes must send messages at a high rate to achieve low algorithm latency. Therefore, in a large scale system, a token-site approach is more favorable when the inter-message arrival time is greater than the delay of the communication step and a symmetric approach is more favorable otherwise. The hybrid algorithm allows some processes to operate using one approach while, at the same time, other processes use the other approach, and is able to commute the operation mode of each process in runtime.

A limitation of the hybrid total order algorithm described in [25] is that it assumes the availability of a perfect failure detector and provides only a regular (non-uniform) version of total order, whose properties are depicted in Table 2. As it will be seen, our new algorithm addresses these two limitations.

4 On the Cost of Total Order

Since on geographically large-scale systems, the network delay is one of the most limiting factors on the performance of a total order algorithm, in this paper we concentrate on analyzing the cost in terms of the number of communication steps needed to provide the service. This measure is less ambiguous than the usual number of "phases". To give an example, the classical two phase commit protocol (2PC) has three communication steps [3]: (1) *vote request* sent from the coordinator to the participants, (2) *reply* of the participants sent to the coordinator, and (3) *decision* sent by the coordinator to the participants. We consider in our analysis only the best case scenario, i.e. runs with no failure suspicions. This is the most frequent case in practice. Using this metric, we now analyze the cost of the several basic total order algorithms and of related building blocks. These

Primitives	*Failure-detector*	*Cost*
Regular reliable broadcast	$\Diamond\mathcal{S}$	1
Non-Uniform Hybrid Total Order	$\mathcal{P}$	min(1+ δ_s, 2)
Uniform reliable broadcast	$\Diamond\mathcal{S}$	2
Uniform Hybrid Total Order	$\mathcal{P}$	min(2+ δ_s, 3)
Consensus	$\Diamond\mathcal{S}$	2
Consensus based total order	$\Diamond\mathcal{S}$	3

Table 3: Communication Steps

costs are summarized in Table 3.

The cheaper of the listed primitives is reliable broadcast, which can be implemented in one communication step. The non-uniform hybrid algorithm exhibits a latency of two communication steps or one-communication step plus δ_s, whichever is more favorable (δ_s is the longest inter-message transmission time of all sequencers) but assumes a perfect failure detector. A uniform reliable broadcast primitive can be implemented in two communication steps [16]. The hybrid protocol can also be trivially extended to support uniform delivery if a perfect failure detector continues to be assumed, by resorting to an additional round of message exchange. The cost of consensus is two communication steps (this is a lower bound [18]). It should be noted that some consensus algorithms require more communication steps even in failure-free runs [5]. In certain special cases, when all processes propose exactly the same value, it is possible to reach consensus in a single communication step. Finally, consensus-based total order algorithms require at least three communication steps (one step to disseminate the message plus the cost of consensus).

In this paper we present a new algorithm that has the following interesting features:

- In stable periods it provides an *UTO-opt-deliver* indication as fast as the non-uniform hybrid total order protocol.
- In stable periods it provides an *UTO-deliver* indication as fast as a hybrid total order protocol extended to ensure uniformity[2].
- In opposition to the hybrid total order protocol (and other protocols of the same class such as typical token-site or symmetric protocols), it does not assume a perfect failure detector. Instead, it never violates safety if failure detection is imperfect

[2]Note that in face of high load and high network delays, the hybrid algorithm offers a much more favorable latency than a consensus based algorithm.

and relies on unreliable failure detectors of the class $\Diamond\mathcal{S}$ for termination.

To motivate the need to design a new algorithm, we first show why some *naive* combinations of the previous approaches do not provide satisfactory results. For instance:

- One might consider to execute the hybrid total-order algorithm (to provide the optimistic delivery) in parallel with a consensus based algorithm (to provide uniform delivery). Unfortunately, it is extremely unlikely that two separate algorithms provide the same ordering. Therefore, in such configuration, the information provided by *UTO-opt-deliver* would be of no practical use.
- Another simple alternative would be to execute the two algorithms sequentially (by using the output of the hybrid algorithm as an input to the consensus based algorithm). However, the total number of communication steps for UTO-deliver in such approach would be too high.

Additionally, it must be noted that the hybrid algorithm, as any other algorithm that assumes a perfect failure detector, needs to be reconfigured when failures are detected. For instance, in a pure token-based approach, when the token holder fails and a new token holder has to be elected. Ensuring that this sort of reconfiguration is still possible even when failure detection is not perfect is a challenging issue addressed by our approach.

5 The Algorithm

5.1 Overview

In the steady-state, the algorithm works closely to the algorithm of [25] with additional steps to ensure uniform total order. Central to the execution of the algorithm is the notion of a *configuration*. A configuration defines which processes assume an active or a passive role with regard to message ordering. Each configuration has an unique configuration id. A new configuration is installed using an underlying consensus algorithm. Each execution of consensus, identified by the associated configuration sequence number, installs a new configuration.

Active processes issue sequence numbers for their own messages and on behalf of (some) passive nodes. The sequence number assigned to a given message is called a *ticket*. Passive nodes do not

issue tickets. In each configuration, each passive node is assigned to an unique active node, called the passive node's *sequencer*. Tickets issued by active nodes are ordered using a symmetric total order algorithm and the associated messages are delivered according to this order. Since the symmetric total order algorithm requires a perfect failure detection to terminate, it may block in case of the failure of an active node. To prevent blocking, when an active node is suspected, a consensus-based phase is used to terminate the algorithm and install a new configuration. A difficulty of the approach is to ensure that the order established by the consensus-based phase never conflics with the order established in steady-state.

Tickets are disseminated in two communication steps. In the first step, the ticket is used to establish an optimistic ordering for the associated message. In the second step, the uniformity of the ticket reception is guaranteed. At the end of the second step of the ticket dissemination, if no failures occur, the associated message can be uniformly delivered. The next paragraphs present the algorithm in greater detail.

5.2 State Variables

The algorithm requires several variables to be maintained at each node (depicted in Figure 2). The set of messages that have been UTO-delivered is stored in variable *u-delivered*. Similarly, the set of messages that have been UTO-opt-delivered is stored in variable *opt-delivered*. In order to be delivered, both the message and its associated ticket must have been previously received. When a message (resp. a ticket) is received for the first time it is stored in variable *r-received* (resp. *r-ticket*). When the uniformity of the message (resp. a ticket) is guaranteed, it is moved to the variable *u-received* (resp. *u-ticket*). The sequence number of the installed configuration is kept in variable *config-sn*. The configuration itself is stored in the variable *curr-config* as a set of tuples (p, s_p). Each tuple associates each process $p \in \Omega$ with a sequencer $s_p \in \Omega$. Active processes are those assigned as sequencers of (at least) their own messages; the initial configuration is pre-defined (usually, a single process is initiated as active). Finally, each process stores the list of suspected processes (*suspected*) and two control variables (*blocked* and *reconfig*) whose purpose will be introduced later in the text.

Initial values:
u-delivered $\leftarrow \emptyset$; // uniformly ordered
opt-delivered $\leftarrow \emptyset$; // optimistically ordered
r-received $\leftarrow \emptyset$; // reliably received
u-received $\leftarrow \emptyset$; // uniformly received
r-ticket $\leftarrow \emptyset$; // reliably received ticket
u-ticket $\emptyset$; // uniformly received ticket
config-sn $\leftarrow 0$
curr-config $\leftarrow \{(p_1, p_1), (p_2, p_1), \ldots\}$
local-sn $\leftarrow 0$; // local sequence number (to issue tickets)
suspected $\leftarrow \emptyset$; // suspected nodes
blocked $\leftarrow$ false;
reconfig $\leftarrow$ false;

Figure 2: State Variables

5.3 Steady-State Operation

The steady-state operation of the algorithm is depicted in Figures 3 and 4. The algorithm is initiated by a request from the application to UTO-broadcast a message. In response to this request, the message is sent to the other processes using an underlying reliable broadcast primitive. Then, if the sender is an active process, it immediately issues a ticket to the message, which is also reliably broadcast (in the implementation, the ticket is piggybacked to the message in order to optimize network resources). Both the message and the ticket are stamped with the identifier of the configuration they are sent in.

When a message is received, it is stored in the *r-received* variable. Additionally, the message is retransmitted to ensure uniformity. If the message was sent by a passive node and is received by that node's sequencer in the same configuration it was sent, the sequencer issues a ticket for the message. Note that messages received in a configuration different from the one they were sent in are not ordered using tickets, they are ordered using a consensus based-algorithm. This may happen when a sequencer is suspected an a new configuration installed. Due to the asynchrony of the system, the sequencer may send several messages with an old configuration before it receives the outcome of consensus.

Tickets are only processed if they are received in the configuration they were sent. In this case, they are saved in the *r-ticket* set and later used to order messages optimistically. Tickets used this way are positively acknowledged using an AckTicket message. Otherwise, tickets are simply discarded and negatively acknowledged using a NackTicket message. When received, the AckTickets for message m (resp. NackTickets) are stored in an $ack\text{-}ticket_m$

variable (resp. *nack-ticket*$_m$). Only tickets that are positively acknowledged from a majority of processes are used to uniformly order messages.

Data messages are moved from the *r-received* set to the *u-received* set as soon as a retransmission is received from a majority of processes. Similarly, tickets are moved from *r-ticket* to *u-ticket* when an AckTicket is received from a majority of processes. If a ticket cannot be moved to the uniform set due to lack of enough positive acknowledgements, the respective message has to be ordered using a consensus-based algorithm. Such algorithm is triggered by setting the *reconfig* flag. There are other three scenarios that may trigger a reconfiguration: *i)* the existence of an (unordered) uniformly delivered message sent in a previous configuration; *ii)* the existence of an (unordered) uniformly delivered message sent in the current configuration but whose sequencer is suspected or; *iii)* the change of the network delays or traffic load leads the adaptation policy to trigger a change of roles (transitions from active to passive and vice-versa). Processes that are suspected are simply stored in the *suspected* variable.

Received tickets (in *r-ticket*) are used to UTO-opt-deliver messages. Similarly, tickets whose uniformity has been guaranteed (in *u-ticket*) are used to UTO-deliver messages. The function next returns the next message that has been ordered and not delivered (if any). More specifically, consider two tickets $T_1 = (\text{Ticket}, p_1, csn, m_1, sn_1)$ and $T_2 = (\text{Ticket}, p_2, csn, m_2, sn_2)$. We say that $T_1 < T_2$ iff, $sn_1 < sn_2 \vee (sn_1 = sn_2 \wedge p_1 < p_2)$. Note that only tickets from the same configuration are comparable. Function next has three parameters: *ret*, a set of received messages; *tset*, a set of tickets and; *dset*, a set of delivered messages. It returns the message $m \in rset$ such that $m \notin dset \wedge \nexists_{m' \in rset} : m' \notin dset \wedge T_{m'} < T_m$ and for every other sequencer s, $\exists_{T_s \in tset} : T_m < T_s$.

5.4 Reconfiguration

Reconfiguration is always performed through the execution of a consensus algorithm. The reconfiguration has two main purposes: to install a new set of roles (active, passive, and sequencer assignments) and to order the remaining unordered messages from the previous configuration.

A reconfiguration is triggered when the flag *reconfig* is activated. The first step consists in collecting the state from other nodes about the list of ordered messages. This is performed by exchanging MyState messages. The MyState message carries the contents of the

```
upon ¬ blocked ∧ UTO-broadcast(m) at p:
    r-received ← r-received ∪{m};
    ack_m ← {p};
    R-broadcast (DATA, p, config-sn, m);
    if (p = sequencer (p)) then
        local-sn ← local-sn +1;
        ack-ticket_m ← ack-ticket_m ∪ {p};
        R-broadcast (TICKET, p, config-sn, m, local-sn);
    fi

upon R-deliver (DATA, q, csn, m) at p:
    ack_m ← ack_m ∪ {q};
    if m ∉ r-received then
        r-received ← r-received ∪{m};
        ack_m ← ack_m ∪ {p};
        R-broadcast (DATA, q, csn, m);
        if (csn = config-sn ∧ p = sequencer (q)) then
            local-sn ← local-sn +1;
            ack-ticket_m ← ack-ticket_m ∪ {p};
            R-broadcast (TICKET, p, csn, m, local-sn);
        fi
    fi

upon ¬ blocked ∧ R-deliver (T ← TICKET, q, csn, m, sn) at p:
    r-received ← r-received ∪{m};
    ack_m ← ack_m ∪ {q, p};
    if (csn = config-sn) then
        r-ticket ← r-ticket ∪{T};
        R-broadcast (ACKTICKET, p, m, T);
        ack-ticket_m ← ack-ticket_m ∪ {p, q};
    else
        R-broadcast (NACKTICKET, p, m, T);
        nack-ticket_m ← nack-ticket_m ∪ {p};
        ack-ticket_m ← ack-ticket_m ∪ {q};
    fi

upon R-deliver (ACKTICKET, q, m, T) at p:
    r-received ← r-received ∪{m};
    ack_m ← ack_m ∪ {p, q};
    r-ticket ← r-ticket ∪{T};
    ack-ticket_m ← ack-ticket_m ∪ {q};

upon R-deliver (NACKTICKET, q, m, T) at p:
    r-received ← r-received ∪{m};
    ack_m ← ack_m ∪ {p, q};
    nack-ticket_m ← nack-ticket_m ∪ {q};
```

Figure 3: Algorithm (Sending and Receiving)

```
upon suspect(q) at p: //
    suspected ← ∪{q};

upon #ack_m > |Ω|/2:
    r-received ← (r-received \{m});
    u-received ← (u-received ∪{m});

upon (#ack-ticket_m + #nack-ticket_m) > |Ω|/2:
    if #ack-ticket_m > |Ω|/2 then
        T ← t ∈ r-ticket: t.m = m;
        r-ticket ← r-ticket \{T};
        u-ticket ← u-ticket ∪{T};
    else if m ∉ u-delivered
        reconfig← true;
    fi

upon next (r-received,r-ticket,opt-delivered):
    m ← next (r-received,r-ticket,opt-delivered);
    opt-delivered ← opt-delivered∪{m};
    UTO-opt-deliver (m);

upon next (u-received,u-ticket,u-delivered):
    m ← next (u-received,u-ticket,u-delivered);
    u-delivered ← u-delivered∪{m};
    UTO-deliver (m);

upon ∃_m ∈ u-received: csn(m) ≠ config-sn):
      ∨ ∃_m ∈ u-received: csn(m) = curr-config.sn ∧ sequencer(m) ∈ suspected
      ∨ performance-reconfiguration:
    reconfig← true;
```

Figure 4: Algorithm (Delivering Messages)

following variables at each node: *suspected*, *u-delivered*, *r-received*, *u-received*, *r-ticket*, *u-ticket*. Basically, it contains the list of all delivered messages, all received messages and all received tickets.

A majority of MyState messages has to be received to compute the new state after reconfiguration. The new state is computed in the following manner:

- The sequence number of the next configuration is simply computed by incrementing the current configuration number.
- The list of suspected members is the union of members suspected by the different processes (this list is only used when assigning the active role to processes: suspected members should be configured as passive nodes). The process excludes from the suspected list all the processes from which is has received a MyState message as an input to the new configuration (including itself).
- All known messages (i.e., those included in the collected MyState messages) are gathered to be delivered before a new configuration is installed.

- All known tickets (i.e., those included in the collected MyState messages) are gathered to be applied before a new configuration is installed.
- Known messages without a ticket are explicitly listed in an *unordered set* (*uset*).
- A new configuration of passive and active roles is constructed using data from the adaptive policy (see discussion below) and the list of suspected members. As noted before, suspected members can only be assigned a passive role.

The new state computed this way is used as an input to consensus. Note that different processes may have used a different set of MyState messages when computing the proposal for the next state. Consensus ensures that all correct processes decide on the same configuration.

The decided configuration is used to update the local state at each node. The set of know messages is added to the *u-received* set (and removed from *r-received*). Similarly, the set of known tickets is added to *u-ticket* (and removed from *r-ticket*). Then, all messages with a ticket in *u-ticket* that have not been delivered are delivered in the order of their tickets. Finally, the list of messages in *u-received* without tickets is delivered in some deterministic order. Note that this particular set of messages is ordered as in the consensus-based total order algorithm of [5]. Finally, the new configuration is installed.

Before initiating a reconfiguration (i.e, before sending its MyState message) each process enters in the blocked state. In blocked state, the process does not send new messages and does not accept new tickets. This prevents the order established by consensus to conflict with an order established using the new tickets. It also minimizes the number of messages sent to an obsolete configuration. The process changes to the unblocked state after installing a new configuration.

5.5 Adaptive Policy

To allow dynamic reconfiguration processes must be able to evaluate system parameters as traffic load and network delays. The following approach may be used: each process timestamps every message with its own local clock at the time of transmission; based on the message's timestamp, all processes can determine the average transmission rate of the sender process. To determine delays in

```
function captureState:
    my-state.csn ← config-sn;
    my-state.susp ← suspected;
    my-state.del ← u-delivered;
    my-state.rec ← r-received ∪ u-received;
    my-state.ticket ← r-ticket ∪ u-ticket;
    return my-state;

function computeNewState (rec-state):
    new-state.csn ← config-sn +1;
    new-state.susp ← ⋃_{s∈rec-state} s.susp;
    new-state.rec ← ⋃_{s∈rec-state} s.del ∪ s.rec;
    new-state.ticket ← ⋃_{s∈rec-state} s.ticket;
    new-state.uset ← new-state.rec \ new-state.ticket ;
    new-state.config ← reassignSequencers (new-state);
    return new-state;

function ApplyState (new-state):
    u-received ← u-received ∪ new-state.rec;
    r-received ← r-received \ u-received;
    u-ticket ← u-ticket ∪ new-state.ticket
    r-ticket ← r-ticket\u-ticket;
    deliverInOrderTickets (u-ticket, u-delivered);
    deliverInDeterministicOrder (new-state.uset);
    config-sn ← new-state.csn;
    curr-config ← new-state.config;
    suspected ← ∅;
    blocked← false;
    reconfig ← false;

upon ¬ blocked ∧ reconfig at p:
    my-state ← captureState;
    R-broadcast (MyState, p, my-state);
    rec-state_csn ← ∪{ my-state});
    blocked← true;

upon R-deliver (MyState, p, s);
    rec-state_s.csn ← ∪{s};
    if ¬ blocked ∧ s.csn ≥ config-sn
        my-state ← captureState;
        R-broadcast (MyState, p, my-state);
        rec-state_config-sn ← ∪{ my-state};
        blocked← true;
    fi

upon # rec-state_config-sn > |Ω|/2:
    proposed-state ← computeNewState (rec-state_csn);
    next-state ← consensus (config-sn+1, proposed-state);
    applyState (next-state);
```

Figure 5: Algorithm (Reconfiguring)

inter-process links, a simple round-trip delay method is used. At every pre-determined fixed interval of time, all receiving processes of a given data message respond immediately with a point-to-point null message to the originator process of the first message. This process can then calculate the delay between itself and all recipients.

In order to evaluate system parameters based on sample measurements, a simple *mean-shift* detector is used: an initial mean value of rate and delay is calculated using the first k samples from each process. Whenever a run of k or more samples fall either all above the mean value or all below it, that mean value is recalculated and used in the next iteration. As the symmetric algorithm relies on the fact that all processes must be constantly sending messages, system parameters can be evaluated after a short period of operation.

With the system parameters the algorithm must assign roles to each process. In order to configure the system, a heuristic that analyses each pair of processes in isolation is used. Consider a process p, subject to a load characterized by a mean inter-message transmission time δ_p, and such that the delay to the nearest (in terms of network delay) active process a is $D_{(p,a)}$. The condition that must be satisfied for process p to assume a passive role is $D_{(p,a)} + \delta_p > 2D_{(p,a)}$. In this case, inter-message transmission time is longer than the round-trip delay to the nearest active process (therefore, p can request and obtain a ticket from a before there is a new message to be sent). On the other hand, if $D_{(p,a)} + \delta_p \leq 2D_{(p,a)}$ then, p should assume an active role since it is sending messages faster than the time required to obtain a ticket from the token-site.

5.6 Correctness

The correctness proofs are presented in Appendix.

6 Performance Evaluation

6.1 Analytical Evaluation

In this section the performance of the presented algorithm is evaluated and compared with the non-uniform hybrid algorithm and the consensus based total order, using communication steps. Let δ_s be the maximum inter-message time of all sequencers of a given configuration. When the sender is the sequencer of its own messages the algorithm requires two communication steps plus δ_s (one for sending the message with the associated ticket, another guaranteeing the uniformity, and δ_s to stabilize the ticket); when the sequencer is a

different process then the algorithm requires three communication steps plus δ_s (one for sending the message, other for sending the ticket, another assuring uniformity, and δ_s to stabilize the ticket). It is important to notice that, due to the adaptive policy of the algorithm, the execution that requires three communication steps is only configured when the latency cost of one communication step is smaller than δ_s. The other important characteristic of this algorithm is that when it makes an optimistic delivery of the messages, this order is equal to the final order when none of the processes is suspected. This delivery is made one communication step before the termination of the broadcast, so the cost of optimistic delivery is one or two, whichever is best in terms of latency. The performance of optimistic delivery is equal to the performance with the non-uniform hybrid total order, presented in the Table 3, so if an application can take advantage of this early delivery the cost of the uniform and non-uniform algorithms is the same. A consensus based total order requires three communication steps, so our algorithm has the same or lower cost while providing the optimistic delivery in the same number of steps as the hybrid algorithm.

6.2 Experimental Evaluation

We have implemented a prototype of our algorithm and performed some simple proof-of concept experiments. Before presenting the collected results, we briefly describe the characteristics of the implementation and the experimental setup.

6.2.1 The prototype

A prototype of the algorithm has been implemented in Java using *Appia*[22], a framework for the composition of micro-protocols. The prototype has various optimizations but these do not change the structure of the presented algorithm. These optimizations are made to reduce the number of messages exchanged by the algorithm and may increase the latency of the message delivery. The optimizations are the following: The *Ticket* and *AckTicket* messages are piggybacked with data messages; the retransmission of messages by processes other than the sender (for reliability) is delayed until the sender is suspected; unless nodes are suspected, message uniformity is obtained through the exchange of acknowledgements (instead of the retransmission of the data message itself).

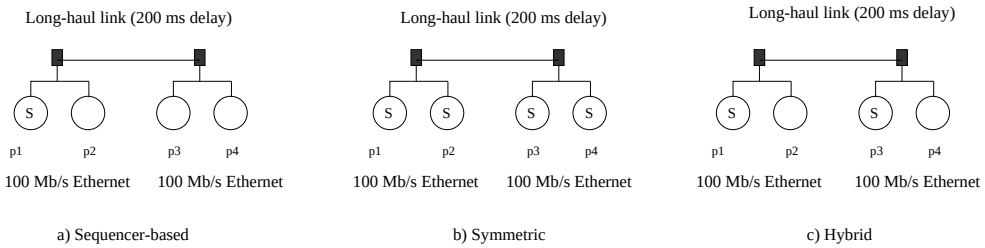

Figure 6: Protocol configurations.

6.2.2 Experience I

The experimental setup consisted of a simple network of four nodes interconnected in a topology consisting of two local-area networks interconnect by a long-haul link. Each node is a PC, equipped with Pentium 3 at 800 Mhz processors with 512M of RAM. In each local-area network, one machine runs the Linux OS and other Windows 2000; all machines run the Java virtual machine version 1.4. The local area networks are 100 Mb/s Ethernets. The long-haul link was simulated using a layer that introduced a random delay with 200 ms average (this value was obtained by measuring round-trip delays from nodes in our lab to different machines in the USA). All nodes were subject to a periodic load. Nodes 1 and 3 have a inter-message transmission time of $100ms$ and nodes 2 and 4 have a inter-message transmission time of $250ms$.

With this configuration, we have measured the performance of three different protocol configurations, as illustrated in Figure 6: *a)* a single sequencer (as in pure sequencer-based approaches); *b)* all processes are sequencers (as in pure symmetric approaches); *c)* the hybrid configuration, with a sequencer in each LAN. All the tests were made in runs where the network conditions were stable and no crashes or failure suspicions occurred. An implementation of the original non-uniform hybrid algorithm, implemented using the same framework, was used as a comparative term. Three values have been measured: the latency of optimistic delivery (*UTO-opt-deliver*), the latency of the definitive delivery (*UTO-deliver*) and the latency of the original hybrid algorithm (*TO-deliver*).

Table 4 shows the average latency of each primitive. We have measured the time elapsed between the transmission of a message and its delivery at the sending node. The values presented are the average of 3 runs where every node transmits 5 consecutive batches of 500 messages. The table presents the results for the values measured in a particular node ($p3$). Note than, in this topology, a single

	TO	UTO-opt	UTO
One-sequencer	475.0	479.0	652.7
Symmetric	328.0	328.0	484.3
Hybrid	303.0	303.0	496.7

Table 4: Performance Results: p3 (ms)

communication step across the long-haul link takes $200ms$.

In the configuration with a single sequencer, the analytical expected latency for node 3 is at least two times the network delay (that is, at least $400ms$, since the sequencer is in the remote LAN). The higher measured value ($475ms$) is due to the fact that, in the prototype, tickets are not sent immediately but piggybacked in the next transmitted message. The difference among the original non-uniform implementation and the optimistic delivery of the indulgent implementation is due to the overhead introduced by the need to exchange acknowledgements to ensure the uniformity of delivery. The implementation also piggybacks the acknowledgements required to achieve uniformity in the normal data traffic. This delays the uniform delivery, which requires $652.7ms$, but reduces considerably the message overhead of the algorithm. It should be noted that a consensus-based algorithm would require at least three communication steps (see Table 3). A similar analysis can be applied for the pure symmetric approach. The original non-uniform algorithm has a latency of $328ms$ which matches the analytical expected value of $(1+\delta)$ communication steps. In this case, the overhead of the indulgent algorithm is not noticeable and, naturally, the uniform delivery exhibits an additional delay that roughly matches the required additional communication step. It is interesting to notice that, in the symmetric approach, the performance is limited by the larger inter-arrival time of processes 2 and 4. For this load configuration, the best performance is achieved by the hybrid approach, assigning a sequencer to each of the processes with smaller inter-arrival time.

6.2.3 Experience II

In order to test the algorithm in more realistic conditions a experiment was made using two local networks, one in Lisbon, Portugal and the other Valencia, Spain. Each node was a PC equipped with Pentium 3 processors with 512MB of RAM, in Portugal, and 256MB in Spain. All nodes were running Linux and the Java Virtual Machine 1.4. The local networks were 100 Mb/s Ethernets and the measured one-way delays in the long-haul link was $53.2ms$.

As presented in the previous experience three different protocol configurations were tested: *a)* Single sequencer; *b)* All processes are sequencers; *c)* Hybrid configuration, with one sequencer in each LAN. The inter-message transmission times were $25ms$ for node *p1* and node *p3* and $100ms$ for node *p2* and node *p4*. All the tests were made when the network conditions were stable and no crashes and failure suspicions occurred. Three values have been measured: the latency of optimistic delivery (*UTO-opt-deliver*), the latency of the definitive delivery (*UTO-deliver*) and the latency of the original hybrid algorithm (*TO-deliver*).

	TO	UTO-opt	UTO
p1	132.2	130.4	179.8
p2	140.5	141.6	192.7
p3	147.1	148.4	181.9
p4	145.6	148.8	190.9

Table 5: Symmetric - Performance Results (ms)

	TO	UTO-opt	UTO
p1	3.5	3.9	147.5
p2	23.3	27.0	213.5
p3	126.1	126.0	219.8
p4	118.7	119.8	227.4

Table 6: One Sequencer - Performance Results (ms)

	TO	UTO-opt	UTO
p1	73.1	76.4	124.1
p2	76.2	76.7	149.3
p3	70.3	70.3	123.3
p4	75.3	88.2	169.8

Table 7: Hybrid - Performance Results (ms)

As in the other experience 3 runs of 5 consecutive batches of 500 messages were measured and the average latency of each primitive in three different scenarios are presented in table5, table6 and table7.

In the configuration with a single sequencer and using the optimistic delivery or the original total hybrid the latency of the sequencer node(*p1*) and the node in the same local network(*p2*) are very small due to the low latency in the local networks and the immediate self-delivery in the sequencer node. In node *p3* and node *p4*

the delay is caused by the two communication steps over a long-haul link (aprox. 100 ms) and the inter-message rate between messages in the sequencer node (25 ms). Using the uniform total order primitive, an extra communication step is needed to assure uniformity, furthermore the use of piggybacking delays the transmission of messages.

In the symmetric configuration using the optimistic delivery or the original total hybrid protocol the latency depend on the largest message transmission cost($50ms$) and the largest inter-message transmission rate($100ms$) Using a uniform total order protocol the cost depends on the sending of the message and the confirmation of a majority of nodes, so the analytical expected latency was $200ms$, $100ms$ due to the cost of message transmission and $100ms$ due to the inter-message transmission time.

In the hybrid scenario the optimistic delivery and the original total hybrid protocol depend on the communication cost and the inter-message transmission rate of the slowest sequencer node. So in this case the maximum cost should be $75ms$; as expected, lower than the average of the other scenarios. For the uniform total order primitive the cost is lower than the other scenarios mainly because the latency between the closest sequencer is also smaller. The analytical expected latency is $200ms$ for the sequencer nodes and $225ms$ for the other nodes.

6.2.4 Summary of evaluation

The collected experimental data confirms the analytical measures. The data also highlights a common tradeoff in this sort of protocols, where a smaller message overhead may be achieved at the cost of an increase of latency: the prototype exhibits a higher latency than the analytical value in all phases where the strategy of piggybacking control messages on data traffic is used in order to reduce the number of messages exchanged by the protocol.

7 Related Work

Optimistic approaches have lately been applied in communication algorithms, specially in total order algorithms. In [23] an atomic broadcast algorithm that extends the Chandra-Toueg algorithm [5] is presented. This algorithm assumes that the physical and data-link layers of local area networks totally order messages in most runs, and that this order can be used as input for a optimistic consensus. The optimistic consensus is an adaptation of the consensus problem

that can reach two results, optimistic and conservative. These conclusions may differ, but the conservative is always the final decision. The optimistic decision is reached if all processes use the same order as input to the optimistic consensus. This algorithm is of limited use in large-scale networks.

A replicated system that uses optimistic algorithms as been presented in [10]. This system uses an optimistic version of a non-uniform sequencer based total order algorithm. The algorithm is optimized for crash-free runs and uses a consensus algorithm when failures are detected. Requests are made by sending a total order message to the group of servers. This total order message is not uniform and the order is not guaranteed in the presence of crashes, but if a majority of processes delivers the message with a certain order then this order can never be changed. When the request is processed by a server a reply is sent to the client, but the response can only be delivered after a majority of replies have been received. This total order algorithm is tailored to this particular form of client-server interaction and difficult to adapt to more general problems.

The Paxos algorithm [21] is a total order algorithm that does not depend on reliable failure detection and that incorporates a lease mechanism. Such mechanism can be seen as a way to optimistically configure the system such that a single process acts as a sequencer for a given batch of messages.

As noted, even if a perfect failure detector is used, uniform delivery is more expensive than non-uniform total order. In fact, depending on the strategy used, some algorithms may even offer worse latency than the best case depicted in Table 3. For instance, in [2] total order is implemented using a token that circulates around the logical ring with the current total order sequence number. In order to guarantee uniformity the token must circulate twice for each message (thus, $2|\Omega|$ communication steps may be needed to order a message).

8 Conclusions

In this paper we have presented an new integrated uniform total order broadcast algorithm that is able to offer optimistic delivery with an efficiency comparable with the best non-uniform algorithms. The optimistic delivery is later confirmed by an uniform delivery indication which is as efficient as any consensus-based total order algorithm. The algorithm has the interesting feature of using local clocks to optimize the system configuration in steady-state. How-

ever, the accuracy of these clocks is not mandatory to ensure the safety of the algorithm outcome. When nodes are suspected, a consensus is executed to order pending messages and reconfigure the system. Therefore, the algorithm combines the best of both worlds: *i)* it offers an early estimate of the definitive order with the efficiency of the algorithms that require the use of a perfect failure detection; and *ii)* the safety of algorithms that make no other assumption than the availability of an asynchronous system augmented with an unreliable failure detector. The algorithm is being used to support the replication of a transactional persistent object repository in geographically large-scale systems [27].

Acknowledgements

The authors are grateful to A. Casimiro, J. Martins and J. Pereira for their comments to earlier versions of this paper.

References

[1] Special issue on group communication. *Communications of the ACM*, 39(4):50–97, 1996.

[2] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison-Welsey, 1987.

[4] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[5] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 34(1):225–267, 1996.

[6] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3), August 1984.

[7] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 544–553, Toulouse, France, June 1993.

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

[9] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In *IEEE 15th Intl. Conf. Distributed Computing Systems*, pages 296–306, May 1995.

[10] P. Felber and A. Schiper. Optimistic active replication. In *Proc. of 21st International Conference on Distributed Computing Systems (ICDCS'2001)*, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.

[11] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. *Pacific Rim Int. Conference on Fault-Tolerant Systems*, December 1995.

[12] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32:374–382, April 1985.

[13] U. Fritzke Jr., P. Ingels, A. Moustefaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, pages 228–234, West Lafayette (IN), October 1998.

[14] R. Guerraoui. Indulgent algorithms. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC'00)*, July 2000.

[15] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[16] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press, 1993.

[17] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. of the 11th International Conference on Distributed Computing Systems*, pages 222–230, 1991.

[18] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults - a tutorial. Technical report, MIT Laboratory for Computer Science, 2001.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. In *Proc. of the Ninth Annual ACM Symposium of Principles of Distributed Computing*, pages 43–57, 1990.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*. 1978.

[21] L. Lamport. The part-time parliament. Technical Report 49, Digital Systems Research Center, Palo Alto, California, May 1989. A revised version of the paper was published in ACM Transactions on Computer Systems, Vol. 16, Number 2.

[22] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, April 2001.

[23] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proc. of the 12th International Symposium on Distributed Computing (DISC'98)*, 1998.

[24] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–146, August 1989.

[25] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proc. of the 16th International Conference on Distributed Computing Systems*, pages 503–510, Hong Kong, May 1996.

[26] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proc. of the Seventh International Conference on Computer Communications and Networks (IC3N'98)*, pages 840–847, Lafayette, Louisiana, USA, October 1998.

[27] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GLOBDATA middleware. In *Proc. of the Workshop on Dependable Middleware-Based Systems (Part of Dependable Systems and Networks Conference, DSN 2002)*, Washington D.C., USA, June 2002.

[28] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proc.of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'20)*, pages 288–295, Taipe, Taiwan, April 2000.

A Correctness Argument

We just prove the properties for the uniform total order delivery. The difficulty of our algorithm is to ensure that the order decided as a result of a consensus (to install a new configuration) never conflicts with the order decided during the stable-period.

Lemma 1. *If a message is inserted in u-received at a given process $p \in \Omega$, then m is eventually UTO-delivered at every correct process.*

PROOF. To insert m in *u-received*, a process $p \in \Omega$ must have received a retransmission of m from a majority of processes. Therefore, there is at least one correct process that has the message. This process will send the message to every correct process. This will cause the message to be uniformly received at every correct process. If the sequencer of the message is correct and the configuration does not change, a ticket will be issued and positively acknowledged by all correct processes. If the sequencer of the message fails, it will be eventually suspected and this will trigger a reconfiguration. Assume that at some point, after one or more reconfigurations, message m has been inserted in *u-received* of all correct processes but not yet *u-delivered*. This will trigger a new reconfiguration (see Figure 4). Since any process q that proposes a new reconfiguration must first collect the *u-received* set from a majority of processes, m will be collected by q. In the state information associated with the installation of the new configuration, m will be necessarily ordered (if a ticket is also collected, the ticket is used, otherwise m is added to the *uset*). By definition of consensus, all correct processes will install that configuration. Therefore, all correct processes UTO-deliver m (as part of the state installation procedure). □

Proposition UTO1. *The algorithm satisfies the Uniform Agreement property (UTO1).*

PROOF. Consider *UTO-broadcast*(m) and a process $p \in \Omega$ that has UTO-delivered m. There are two cases that can cause m to be UTO-delivered:

1. m was delivered as part of the agreed state of a reconfiguration process, or;
2. at p, m was uniformly received and $ticket_m$ has been positively acknowledged by a majority of processes.

In the first case, by definition of consensus, all correct processes will also install the same state (as a result of the corresponding

consensus execution), and the message is delivered by all these processes. In the second case, the proof derives directly from Lemma 1. □

Proposition UTO2. *The algorithm satisfies the Termination property (UTO2).*

PROOF. Consider the execution of *UTO-broadcast*(m) by a correct process p. By the property of reliable broadcast, every correct process R-delivers(m). Thus every correct process will retransmit m and m becomes *u-received* at every correct process. The rest of the proof derives from Lemma 1. □

Lemma 2. *Let m be a message that is* UTO-broadcast. *If a process uses a $ticket_m$ to order a message, in a given configuration, every correct process orders the message m using that ticket in the same configuration.*

PROOF. In order to be UTO-delivered at p, message m must have been added to set *u-received* at p and $ticket_m$ must have been positively acknowledged by a majority of processes. This means that $ticket_m$ must have been added to *r-ticket* at a majority of processes.

If there is no reconfiguration, the sequencer for that message will ensure every correct process will receive and acknowledge positively the $ticket_m$. Therefore, every correct process will eventually UTO-deliver m using $ticket_m$.

If there is a reconfiguration, the tickets to be used before a new configuration is installed, as part of the state installation procedure, are the union of the *r-ticket* sets collected from a majority of processes. At least one process q that has acknowledged $ticket_m$ to p must belong to any such majority. Therefore, $ticket_m$ belongs to the state proposed by any process to the next configuration. By definition of consensus, $ticket_m$ will be processed by every correct process before the new configuration is installed. □

Lemma 3. *Let m be a message that is* UTO-broadcast. *If a process $p \in \Omega$ uses the deterministic order of a* uset *to deliver a message, then no process orders this message using $ticket_m$.*

PROOF. Let q be a process that proposes a configuration where m belongs to the *new-state.uset*. For m to be added to *new-state.uset*, q did not receive $ticket_m$ in any of MYSTATE messages. On the other hand, q has waited for at least a majority of MYSTATE messages before proceeding. Therefore, there is at least a majority of processes that did not include $ticket_m$ in their *r-ticket* sets before

they blocked their state. The processes in this majority will only unblock their state when they receive a new configuration and, at that point, refuse further tickets from the previous configuration. This means that no process was or will be able to collect a majority of AckTicket messages required to use $ticket_m$ to order m. □

Proposition UTO3. *The algorithm satisfies the Uniform Total Order property UTO3.*

Proof. Messages are either:

- *i)* UTO-delivered in the order defined by their sequence tickets, or;
- *ii)* UTO-delivered in deterministic order from the *new-state.uset* of a consensus execution.

By Lemma 2, if a message m is delivered by a process $p \in \Omega$ according to $ticket_m$, then every process $q \in \Omega$ delivers m according to $ticket_m$. If a process $p \in \Omega$ orders m in deterministic order from the *new-state.uset* of a consensus execution, by definition of consensus, every correct process will agree on the same *new-state.uset* and deliver the message in the same order. Additionally, by Lemma 3, if a process uses method *ii)*, no process ever uses method *i)*. □

Proposition UTO4. *The algorithm satisfies the Integrity property UTO4.*

Proof. The proof derives trivially from the integrity property of the regular reliable broadcast and from the fact that messages that have been *UTO-delivered* are registered in the *u-delivered* variable and not delivered again. □