Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Group Communication Protocols Report

GORDA Deliverable D3.5

Actual submission date: 2007/09/06

Start date of project: 1 October 2004 Duration: 42 Months

Universidade de Lisboa

Revision 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contributors

José Mocito, U. Lisboa
José Pereira, U. Minho
Lusís Rodrigues, U. Lisboa
Nuno Carvalho, U. Lisboa
Rui Oliveira, U. Minho

(C) 2006 GORDA Consortium. Some rights reserved.
This work is licensed under the Attribution-NonCommercial-NoDerivs 2.5 Creative Commons License. See http://creativecommons.org/licenses/by-nc-nd/2.5/legalcode for details.

Abstract

This document reports on the work that has been performed on selection, development and evaluation of new algorithms to support wide-area and clustering database replication protocols. These algorithms include total order with optimistic assumptions and the definition of primary views on partitionable systems.

This work focused on total order protocols, their evaluation and how can they be dynamically changed adapt to network fluctuations. We also report the work done on how to improve the group communication solution adopted by GORDA to fit the project requirements. The requirements include security on the communication and the definition of management interfaces.

Contents

1 Introduction **2**
1.1 Objectives 2
1.2 Relationship with other deliverables 3

2 Switching algorithm for total order protocols **4**
2.1 Protocol description 6
2.2 Evaluation 7
2.2.1 Switching overhead 7
2.2.2 Comparative analysis 8

3 Improved SETO protocol **11**
3.1 New heuristics to calculate delays 13
3.2 Protocol description 14
3.3 Evaluation 15
3.3.1 Network plane size 15
3.3.2 Process transmission rates 16
3.3.3 Sequencer position 16

4 Improving the Appia toolkit **18**
4.1 Providing primary partitions 18
4.1.1 Protocol overview 19
4.1.2 Allowing recovery 19
4.2 Management interfaces 19
4.3 Notification of services ensured 20
4.4 Communication security 20

Chapter 1

Introduction

The main goal of this deliverable is to present and discuss (i) the group communication toolkit and its configurability and (ii) the new protocols for total order and primary views, needed for clustering and wide-area replication protocols.

Group communication supports message passing within groups of processes by offering membership management and reliable multicast services. Membership management keeps track of which processes are operational and mutually reachable, taking into account both voluntary requests to join and leave the group as well as process failures and network partitions. By ensuring that a common membership is observed by all participants, many distributed algorithms are simplified. Message ordering simplifies application programming by ensuring that each message is handled in a predictable context resulting from previous messages. The definition of primary views helps the recovery processes and avoids that the state of the database replicas diverges.

This document reports the efforts and progress made in the scope of the project to improve the group communication toolkit for clustering and wide-area replication protocols, namely:

- a new total order protocol that combines several implementations and allows that the system can change at run time the instance being used for better performance;
- an improved statistically estimated optimistic total order protocol;
- a protocol that uses a partitionable group communication system and provides to the replication protocols the notion of primary partitions, avoiding state divergence; and
- the efforts made in the group communication toolkit to cope the project needs.

1.1 Objectives

The goals of the work reported in this document are as follows:

- improve the group communication toolkit for the new GORDA database replication protocols;
- design appropriate total order protocols that are adaptive and well suited for clustering and wide-area networks;
- support the notion of primary partitions and ease the recovery process.

1.2 Relationship with other deliverables

This deliverable departs from the work on D1.1 (State of the Art Report) and, to some extent, the choices herein are influenced by D1.2 (User Requirements Report), D1.3 (Strategic Research Directions Report) and D2.3 (GORDA Interfaces Definition). It is instrumental on shaping the work to be delivered with D3.3 (Replication Modules Reference Implementation). The current deliverable has two companion reports: D3.1 (Wide-Area Oriented Protocols Report) and D3.2 (Cluster Oriented Protocols Report).

Chapter 2

Switching algorithm for total order protocols

A total order broadcast protocol is a fundamental building block in the construc- tion of many distributed fault-tolerant applications [18] and in particular in the construction of most of the database replication protocols of the GORDA project. Informally, the purpose of such a protocol is to provide a communication primitive that allows processes to agree on the set of messages they deliver and, also, on their delivery order. Uniform total order broadcast is particularly useful to implement fault-tolerant services by using software-based replication [10]. Unfortunately, the implementation of such a primitive can be expensive both in terms of communication steps and number of messages exchanged. This prob- lem is exacerbated in large-scale systems, where the performance of the algorithm may be limited by the presence of high-latency links. Several total order protocols have been proposed that use different strategies to offer good performance [7]. There is no protocol that outperforms all others in all scenarios: each protocol offers best results under different load profiles and/or network conditions.

Informally, total order broadcast is a group communication primitive that ensures that messages sent to a set of processes are delivered by all those processes in the same order. Such a primitive is useful, for example, in the implementation of fault-tolerant services [18], for instance, using the state machine approach (active replication) [22]. More formally, total order broadcast is defined on a set of processes Ω by the primitives (1) *TO-broadcast*(m) which issues message m to Ω, and (2) *TO-deliver(m)* which is the corresponding delivery of m. When a process p_i executes *TO-broadcast*(m) (resp *TO-deliver(m)*), we say that p_i *"TO-broadcasts m"* (resp *"TO-delivers m"*). The total order primitive characterized by the properties listed in Table 2.1 is known as regular total order. A stronger version, called uniform total order [7], can also be defined. The difference among these definitions is not relevant for understanding our adaptive protocol, thus we will not delve further in this topic.

Many algorithms exist to implement total order. To give an insight on the possible alternatives, we briefly introduce two of the most used ones, namely the *sequencer-site* [11] and the *symmetric* [17, 8] approach. Both methods have advantages and dis-

TO1 - Total order: Let m_1 and m_2 be two messages that are *TO-broadcast*. Let p_i and p_j be any two correct processes that *TO-deliver*(m_1) and *TO-deliver*(m_2). If p_i *TO-delivers*(m_1) before *TO-delivers*(m_2), then p_j *TO-delivers*(m_1) before *TO-delivers*(m_2), and we note $m_1 < m_2$.

TO2 - Agreement: If a correct process in Ω has *TO-delivered*(m), then every correct process in Ω eventually *TO-delivers*(m).

TO3 - Termination: If a correct process *TO-broadcasts*(m), then every correct process in Ω eventually *TO-delivers*(m).

TO4 - Integrity: For any message m, every correct process *TO-delivers*(m) at most once, and only if m was previously *TO-broadcast* by some process $p \in \Omega$.

Table 2.1: Regular total order properties

advantages.

In the sequencer-site approach one site is responsible for ordering messages on behalf of the other processes in the system. Sequencer-based protocols 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 these protocols, a message sent by a process that is not the sequencer experiences a delivery latency close to $2D$, where D is the message transit delay between two system processes (i.e., the time to disseminate the message plus the time to obtain an order number from the sequencer process). Thus, sequencer-based approaches are inefficient in face of large network delays. Note that it is possible to design solutions where the sequencer role is rotated among processes [5].

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* [13] or *vector clocks* [3, 17]: messages are delivered according to their partial order and concurrent messages are totally ordered using some deterministic algorithm. Symmetric protocols have the potential for providing low latency in message delivery when all processes are producing messages. In fact, symmetric protocols can exhibit a latency close to $D + t$, where t is the largest inter-message transmission time [19]. Unfortunately, this also means that all (or at least a majority [8]) processes must send messages at a high rate to achieve low protocol latency.

Several other alternatives exist. For a comprehensive survey, the reader is referred to [7]. However, from the two examples above, it should be clear that it is interesting to have a protocol that can dynamically adapt to changes in the operation envelope by switching, in run-time, from one algorithm to another.

In this chapter we describe and evaluate a total order protocol that combines different algorithms and adapts itself to the running environment. The protocol allows a fluid transition between algorithms, never stopping the flow of application messages.

```
1: Initialization:
2:     deliv ← ∅
3:     undeliv ← ∅
4:     curAlg ← TO-A {current algorithm}
5:     newAlg ← ∅ {next alg.}
6:     switching ← false
7:     check[1..n] ← false

8: upon changeAlgorithm(newTO) do
9:     rBroadcast(switch,newTO)

10: upon rDeliver(switch,newTO) do
11:     newAlg ← newTO
12:     switching ← true
13:     TO-broadcast(curAlg,(flag,null,myself))

14: upon TO-deliver(curAlg,(flag,null,sender)) do
15:     check[sender] ← true

16: upon check[1..n] = true do
17:     endSwitch()

18: upon TO-broadcast(msg) do
19:     TO-broadcast(curAlg,msg)
20:     if switching = true then
21:         TO-broadcast(newAlg,msg)

22: upon TO-deliver(alg,msg) do
23:     if alg = curAlg ∧ msg ∉ deliv then
24:         deliver(msg)
25:         deliv ← deliv ∪ {msg}
26:     else if msg ∉ deliv then
27:         undeliv ← undeliv ∪ {msg}

28: procedure endSwitch()
29:     for all msg ∈ undeliv ∧ msg ∉ deliv do
30:         deliver(msg)
31:         deliv ← deliv ∪ {msg}
32:     undeliv ← ∅
33:     check[i..n] ← false
34:     curAlg ← newAlg
35:     switching ← false
```

Figure 2.1: Adaptive Total Order algorithm

2.1 Protocol description

To be able to effectively transition from one algorithm to the other, all nodes need to agree on the point in the message flow where they switch. Also, both algorithms must provide FIFO ordering of messages (which is the most common case). The rational behind our proposal is to start broadcasting messages using both total order algorithms, during the switching phase, until a safe point is reached in every process. By using both algorithms simultaneously, no stoppage in the message flow is necessary. The protocol is listed in Figure 2.1.

Let us assume that the adaptation protocol is using algorithm TO-A to order messages and wants to switch to algorithm TO-B. The transition protocol works as follows. A control message is broadcast to all processes to initiate the reconfiguration (lines 8–9). When a node receives this message (line 10) it starts broadcasting messages using both total order algorithms. Also, the first message it broadcasts using algorithm

TO-A is flagged. If no message is to be sent, then a flagged special null message is broadcast using TO-A, to allow faster protocol termination (flagged first message is not represented in the algorithm to preserve clarity). When a process starts receiving messages from both TO algorithms it performs the following steps (lines 22–27): messages received from TO-A are delivered as normally; messages received from TO-B are buffered in order. As soon as a flagged message is received from each and every node (line 15) the transition is concluded using the following "sanity" procedure (lines 28–35). Firstly, all messages received from TO-B that have not yet been delivered by TO-A are delivered in order. Finally, from this point on, all messages received from TO-A are simply discarded and no further message is sent using TO-A (until a new reconfiguration is needed). The TO-B algorithm is then used to broadcast and receive all the messages to be delivered.

Note that, after the transition is concluded, messages received from TO-B are delivered only if they have not been already received and delivered from TO-A (line 23). This is a necessary safeguard as the two total order algorithms do not necessarily deliver messages in the same order, nor at the same time. So there is a possibility that a message that has already been delivered from TO-A is received after the termination of the reconfiguration procedure from TO-B.

Also, the protocol presented does not allow concurrent adaptations. For one adaptation to happen, the previous (if any) should always have concluded.

2.2 Evaluation

We evaluate the performance of the adaptive protocol from two different perspectives. First, we evaluate the overhead of the switching procedure. Then, we provide a comparative analysis on how different switching strategies interfere with the traffic flow during the reconfiguration.

2.2.1 Switching overhead

To evaluate the switching overhead of our adaptive protocol we compare the performance of a system that always uses the same total order algorithm, with that of a system that is periodically switching between two algorithms. To make the comparison as fair as possible, we made our protocol switch between two instances of the same total order algorithm, which is also used as the non-adaptive protocol. Also, the network topology and working conditions did not change during the tests. In this way, we can isolate the cost of the switching procedure given that all the remaining factors remain unchanged.

The adaptive protocol was implemented in Java using the Appia toolkit. The experiments were conducted in the SSFNet [15] network simulator and the scenario consists of a five node cluster, where all nodes are connected to each other by 100Mbps bidirectional links.

Two runs of the same experiment were performed: (A) one using a single total order protocol (non-adaptive), (B) and another using the proposed adaptive total order protocol, which is forced to switch periodically. Each run consists of every node

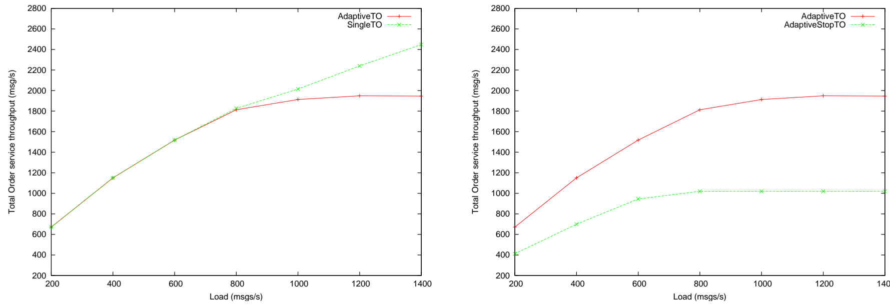


Figure 2.2: TO throughput in non-adaptive, adaptive and optimized algorithms

Figure 2.3: TO throughput in adaptive and stop algorithms

broadcasting 5000 messages of 5KB in total order. The experiment ends when all nodes receive all the broadcast messages. The values presented are averages of the measurements conducted in each node.

Figure 2.2 presents the overall throughput results when the send rate is made variable. As depicted, both total order algorithms perform the same until they reach approximately 800 msg/s. After this point, the throughput of the non-adaptive protocol continues to grow while its value stabilizes for the adaptive protocol. This behavior is explained by the overhead introduced by the switching phase in the adaptive protocol. During this phase, the same set of messages is being broadcast by two total order algorithms at the same time, leading to an increase (approximately double) in the bandwidth usage. If the send rate is too high, the available bandwidth can be exhausted, leading to the stagnation observed in the throughput.

Thus, we can conclude that our switching protocol offers negligible overhead as long as there is enough network bandwidth to support the transmission of data in parallel during the reconfiguration. When the protocol operates close to the available bandwidth, the switching procedure introduces an overhead. This limitation can be addressed at the implementation level, by sending the payload of the messages using just one of the two algorithms. The overhead of this optimization depicted in Figure 2.2.

2.2.2 Comparative analysis

Most switching protocols require the message flow to be stopped in order to terminate the reconfiguration process. By not imposing a gap in the message flow, our protocol provides smooth transitions between algorithms, thus allowing applications that rely in its services to normally execute, even during the switching phase. Therefore, it should offer better overall throughput, as long as enough bandwidth is available to cope with the demand imposed by the transmission of messages using two algorithms at the same time. The same experiment described in 2.2.1 was conducted using a protocol that stops the message flow. This protocol operates by sending a stop request to all nodes and awaiting for a confirmation from each of these nodes. After confirming the stop request a node does not send further messages until the switch is complete.

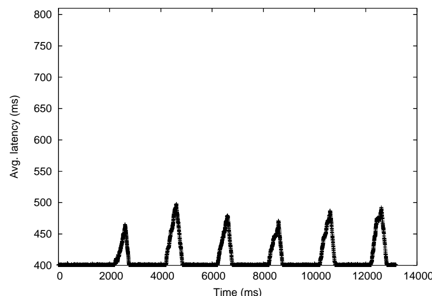


Figure 2.4: Latency in Adaptive TO

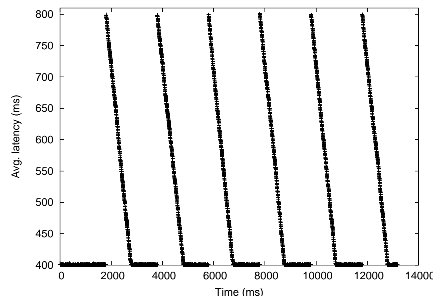


Figure 2.5: Latency in RABP

The performance of such protocol when compared to our proposal can be observed in Figure 2.3, which clearly shows that our approach always performs better.

Other protocols that try to minimize the cost of switching between algorithms have also been proposed. A previous work [20], proposes a solution that has some similarities with our protocol, but differs from it by not requiring every node to wait for a "special" (in our algorithm the term is "flagged") message from every other node, and also for not making any assumptions about the failure model where it is executing. In [20], a special reconfiguration message is broadcast in total order. When a node receives such message, it stops the flow in the current algorithm, and re-issues all his undelivered messages in the next algorithm. It then starts using it to broadcast messages in total order. We will refer to this protocol by RABP (*Replacement of the Atomic Broadcast Protocol*).

The RABP strategy has the advantage of requiring less bandwidth during the switching phase. However, some delay is imposed to the message flow during the retransmission of the undelivered messages. To observe this side effect, the experiment was now conducted using our protocol and the RABP protocol. In Figures 2.4 and 2.5 we can observe how both compare in terms of latency. The spikes depicted correspond to the switching phases, in the time-line of the experiment. The inter-arrival time of messages was also measured and its evolution is shown in Figures 2.6 and 2.7. Finally, the number of messages delivered by a fixed period of time (10 ms) was also observed and the comparative results are depicted in Figures 2.8 and 2.9.

This experiment clearly showed that our proposal is able to keep a sustained delivery rate during the switching phase and performs similarly to RABP during the remaining time. By not significantly delaying the message flow, our protocol can best suit environments where application stoppage, due to significant communication delays, is not desirable.

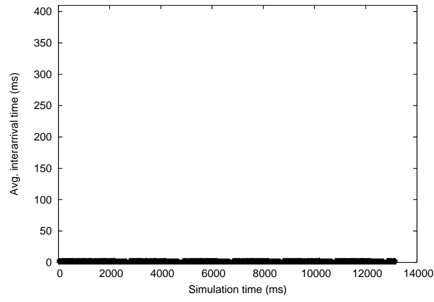


Figure 2.6: Inter-arrival time in Adaptive TO

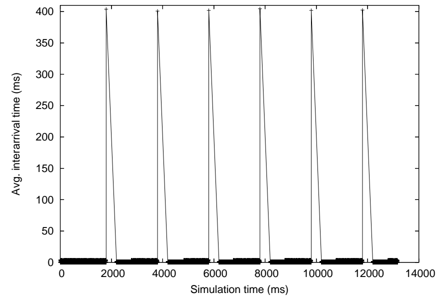


Figure 2.7: Inter-arrival time in RABP

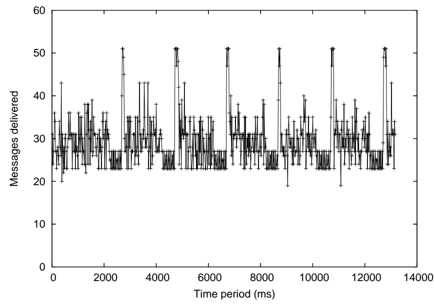


Figure 2.8: Delivery rate in Adaptive TO

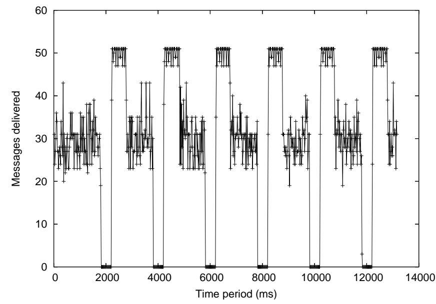


Figure 2.9: Delivery rate in RABP

Chapter 3

Improved SETO protocol

The notion of optimistic total order was first proposed in the context of local-area broadcast networks [16]. In many of such networks, the spontaneous order of message reception is the same in all processes. Moreover, in sequencer-based total order protocols the total order is usually determined by the spontaneous order of message reception in the sequencer process. Based on these two observations a process may estimate the final total order of messages based on its local receiving order and, therefore, provide an optimistic delivery as soon as a message is received from the network. With this optimistic delivery, the application can make some progress. For example, a database replication protocol can apply the changes in the local database without committing it. The commit procedure can only be made when the final order is known and if it matches the optimistic order. If the probability of the optimistic order matching the final order is very high, the latency window of the protocol is reduced and the system gains in performance.

Such approach is unfeasible in large-scale networks. The long latency in wide-area links causes different processes to receive the same message at different points in time. Consider the topology depicted in Figure 3.1. Assume that process a multicasts a message m_1 and that, at the same time, the sequencer s multicasts a message m_2. Clearly, the sequencer will receive $m_2 < m_1$, given that m_1 would require $12ms$ to reach the sequencer. On the other hand, process b will receive $m_1 < m_2$, as m_1 will take only $2ms$ to reach b while m_2 will require $12ms$. From this example, it should be obvious that the spontaneous total order provided by the network at b is not a good estimate of the observed order at the sequencer.

To address the problem above, [23] proposed to introduce artificial delays in the message reception to compensate for the differences in the network delays. It is easier to describe the intuition of the protocol by using a concrete example. Consider again the network of Figure 3.1. Assume also that we are able to provide to each process an estimate of the network topology and of the delays associated with each link. In this case, b could infer that message m_1 would take $10ms$ more to reach s than to reach b. By adding a delay of $10ms$ to all messages received from a, it would mimic the reception order of a's messages at s. A similar reasoning could be applied to messages from other processes.

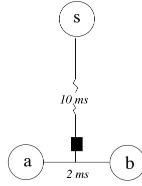


Figure 3.1: Local and wide-area links.

In this chapter we describe a new total order algorithm that improves previous [23] work by making a better estimation of the network delays. We will show that the algorithm used in [23] fails to offer the optimal average latency of optimistic deliveries. Departing from this observation we discuss how the optimal delays can be computed and, subsequently, propose an efficient heuristic to approximate the optimal result in an cost-effective manner.

In this section we seek solutions that minimize the overall average latency (OAL) of all optimistic deliveries, denoted Δ_{avg}. Let r_i be the rate at which process p_i sends messages. Δ_{avg} is defined as:

$$\Delta_{\text{avg}} = \frac{\sum_{i,j=1}^{N} r_i \delta_i^j}{N \sum_{j=1}^{N} r_j} \tag{3.1}$$

Minimizing the overall average latency of all optimistic deliveries is equivalent to Minimize the Overall Latency (MOL) and this problem can be mathematically formulated in linear programming as:

MOL:

$$min \sum_{i,j=1}^{N} r_i \delta_i^j \tag{3.2}$$

s.t.:

$$\delta_i^1 - \delta_{i+1}^1 = \delta_i^j - \delta_{i+1}^j, \quad i = 1, ..., N-1, j = 2, ..., N \tag{3.3}$$

$$\delta_i^j \geq 0, \quad i, j = 1, \ldots, N. \tag{3.4}$$

Equation (3.2) states the objective function to be minimized: the weighted sum of all the latencies. Equation (3.3) ensures that the latencies under decision will give rise to a total order. Finally, equation (3.4) ensures that all the latencies are non-negative.

Let ω_i^j denote the network delay of messages from process p_i to process p_j and let x_i^j denote the delay these messages should suffer to accomplish the corresponding latency δ_i^j, $i, j = 1, ..., N$. Consequently, by rewriting δ_i^j as $\delta_i^j = \omega_i^j + x_i^j$, and considering x_i^j as the new decision variables, the MOL can be reformulated as:

$$min \sum_{i,j=1}^{N} r_i x_i^j \tag{3.5}$$

s.t.:

$$(\omega_i^1 + x_i^1) - (\omega_{i+1}^1 + x_{i+1}^1) = (\omega_i^j + x_i^j) - (\omega_{i+1}^j + x_{i+1}^j), \quad i = 1, ..., N-1, j = 2, ..., N \tag{3.6}$$

$$x_i^j \geq 0, \quad i, j = 1, \ldots, N. \tag{3.7}$$

In this new formulation, equation (3.5) states the objective function to be minimized: the weighted sum of the delays to impose to all the messages. Equation (3.6) ensure total order, while equation (3.7) guarantees for the non negativity of the delays.

3.1 New heuristics to calculate delays

This problem has N^2 decision variables and $(N-1)^2$ constraints of type (3.6). Instances of the MOL problem can be solved using a solver of Linear Programming models, such as ILOG CPLEX [6]. Unfortunately, it may be unpractical to install and execute such solver in each protocol stack of every process of the distributed system. Therefore, in this section we provide a heuristic to calculate the delays to be applied to each incoming message, that approximates the optimal solution. Our heuristic works as follows. We incrementally build a network by adding one process at a time. The first process to be added defines the first message in the total order $\prod$, named π_1. This first process has complete freedom to set the delay it imposes on its own messages (which may be zero).

Whenever another process is added to the network it tries to set itself as close as possible to π_1 in the total order $\prod$, subject to the restrictions from equations (3.6) and (3.7). Note that the later a process is inserted in the network, the larger the set of constraints it has to satisfy; therefore, it is likely that later processes may be required to add longer delays to their own messages.

A key point in the heuristic is the order by which processes are inserted in the network. To define this order, we use the following insight. Consider process p_k. When process p_k is added to the network, the delay that this process has to impose to its own messages x_k^k depends of the constraints imposed by processes $p_1, \ldots, p_{k-1}$ previously inserted. Consider now the next process to be added to the system p_{k+1} and ω_k^{k+1} the latency of the link between p_k and p_{k+1}. If $x_k^k > \omega_k^{k+1}$, p_{k+1} will be forced to impose a delay to its own messages of at least $\omega_k^{k+1} - x_k^k$. This will happen if p_k is very close to p_{k+1} and is forced to impose a long delay to its own messages. This means that the closer a process is to other processes, the more likely it is to influence the delays imposed on the messages from those processes. Thus, processes that are closer to other processes should impose the minimum delays to their own messages. Since processes that are inserted earlier in the network are more likely to impose smaller delays (as they have less constraints to satisfy), these processes should be the ones to be inserted first. The heuristic described in the next paragraphs, uses a precise metric to capture the fuzzy notion of "closeness" introduced here.

The paragraph above explains the negative impact of setting x_k^k such that $x_k^k > \omega_k^{k+1}$. It is interesting to note however, that it may not be always desirable to set

```
1:  Initialization:
2:      𝒫 ← p_1, ..., p_n {Process group}
3:      delay[1..n] ← 0 {Delay applied to messages}
4:      tdelay[1..n] ← 0 {Transmission delays to the process}
5:      c_t delay[1..n][1..n] ← 0 {Complete transmission delay matrix}

6:  procedure computeDelays()
7:      {computes delays using the solver or the heuristic}

8:  upon R_deliver(DELAY(new_delay[]) do
9:      if sender = seq then
10:         c_t delay[new_delay.sender] = new_delay
11:     else
12:         delay = new_delay

13: procedure updateDelays()
14:     R_unicast(seq,DELAY(tdelay))

15: upon allDelaysGathered() do
16:     c_delay ← computeDelays()
17:     for all p_i ∈ 𝒫 do
18:         R_unicast(p_i,DELAY(c_delay[p_i])

19: procedure TO_multicast(m)
20:     R_multicast(DATA(m))

21: upon R_deliver(DATA(m) do
22:     R ← R ∪ {(m, now + delay[m.sender])}

23: upon ∃(m, d, t, md) ∈ R : now ≥ t ∧ m ∉ O ∧ m ∉ F do
24:     opt_delivery(m)
25:     O ← O ∪ m
26:     if p = seq then
27:         g ← g + 1
28:         R_multicast(SEQ(m, g))

29: upon R_deliver(SEQ(m, s)) do
30:     S ← S ∪ {(m, s, now)}

31: upon ∃(m, d, o) ∈ R : (m, l + 1, t) ∈ S ∧ m ∉ F do
32:     fnl_delivery(m)
33:     l ← l + 1
34:     F ← F ∪ {m}
```

Figure 3.2: Fast SETO algorithm.

$x_k^k = 0$. In fact, by setting $x_k^k = 0$, we are forcing p_{k+1} to set $x_{k+1}^{k+1} = \omega_k^{k+1}$. So, there is a trade-off between the delay process p_k imposes on its own messages and the delay that other processes will later have to impose on their own messages.

3.2 Protocol description

We now present an augment version of the SETO algorithm, that we have named Fast SETO, that can work with the heuristic or by calling a solver to obtain the minimum optimistic latency. The complete Fast SETO algorithm is specified in Figure 3.2. The algorithm works in four steps. In the first step, every process collects round-trip delays from all the other processes and estimates the corresponding transmission delays. This step is omitted in the algorithm specification for clarity sake, given there are multiple ways of collecting round-trip estimations and that procedure is orthogonal to

the main algorithm. For instance, if TCP is used as the underlying transport protocol, the round-trip estimation could be extracted from the TCP implementation without any extra cost. In the second step, all processes send the gathered delay information to a specific process. This process's identity can be easily derived from the group membership; for instance, it can be the process with the smallest identifier. The process gathers all the delay information and computes the optimal delay when a solver is available, or approximates the optimal solution using the heuristic described in the previous section when the use of a solver is impractical. Finally, it sends to each process in the group the corresponding line in the delay matrix, which holds the delays that must be enforced by that specific process.

Our algorithm clearly differs from the original SETO algorithm by requiring complete knowledge of the transmission delays between all processes, which translates into the exchange of distance vectors between all nodes. The original SETO requires no such information, making use of only local clock values to determine the artificial delays. However, our proposal significantly improves the overall average latency of the system, as will be shown in the next section.

3.3 Evaluation

In this section we evaluate our proposed algorithm against the optimal delay assignment and the original SETO algorithm. The evaluation tests were performed in a simulated environment that consists of a network topology, transmission rates associated with each node and three models that describe the three algorithms at stake: optimal assignment, original SETO and Fast SETO using the heuristic. The network topologies used were generated with BRITE [14]. The tests were performed in networks with 30 nodes (10 nodes for the last evaluation test) that where randomly placed in a topological space.

3.3.1 Network plane size

We first compare the performance of the optimal assignment, original SETO and Fast SETO when the network plane size is changed. BRITE allows for the definition of the plane size by specifying the dimension of one side. In the experiments performed we made this side vary between 1000 and 5000 units. For each space dimension 20 network topologies were generated, and the results shown are average values of the observations on those networks. Also, in the original SETO algorithm a randomly selected sequencer was used.

The results are depicted in Figure 3.3. The explanation for these results lies in the way the original SETO algorithm determines the delay imposed by the sequencer to its own messages, which is equal to the longest link that reaches the sequencer. This value then conditions the adjustments in the remaining nodes and produces the observed results. The results also show the improvements obtained by Fast SETO in regard to the original algorithm and also its proximity to the optimal solution. In the experiments, the original SETO algorithm was, on average, 66% to 114% worst than the optimal assignment. Fast SETO with the heuristic was, on average, 16% to 33%

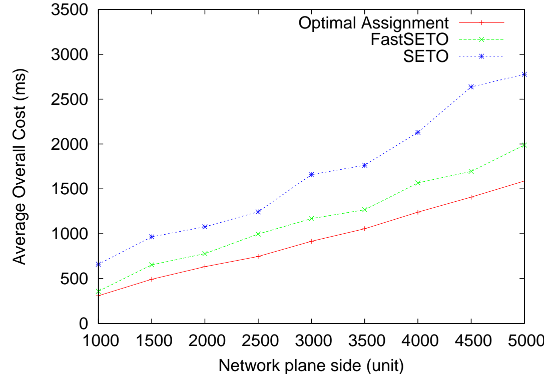


Figure 3.3: Network diameter.

worst than the optimal assignment, which shows the significant gains obtained by using our proposed algorithm.

3.3.2 Process transmission rates

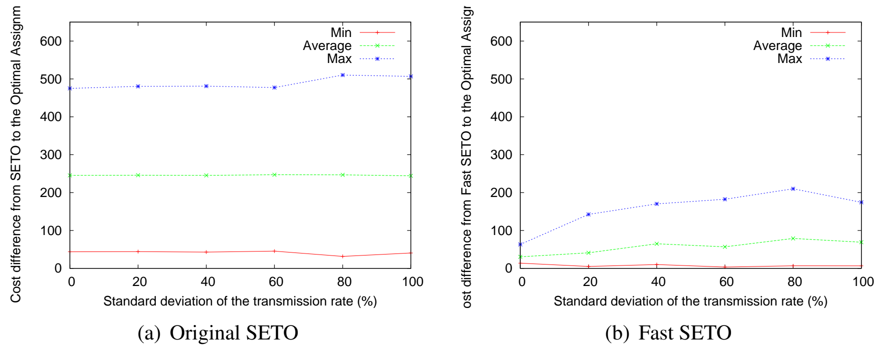
We now compare the performance of the optimal assignment, original SETO and Fast SETO when the transmission rates of the processes in the network are changed. This time we set the topological space to a constant value. Each experiment consisted in generating 20 different topologies where all nodes exhibited an average transmission rate of 300 messages per second, with a predefined variance. The standard deviation of the transmission rates for each experiment was made variable between 0% to 100%. As in the previous experiment, the sequencer for the original SETO algorithm experiments was randomly selected.

Figures 3.4(a) and 3.4(b) hold the results for both the experiments comparing the original SETO with the optimal assignment and Fast SETO with also the optimal assignment, respectively. Each figure presents the results as differences from each algorithm to the optimal assignment. The three lines presented in each figure are the minimum, maximum and average values observed from all the 20 topologies for each standard deviation value.

As expected, the average overall cost of the Fast SETO algorithm suffers less variation than the original SETO algorithm. The reason for this is that Fast SETO takes into account the transmission rates when computing the artificial delays. The original SETO algorithm makes no use of this information, which makes its results more dependent of the specific topology where it is executing.

3.3.3 Sequencer position

The final evaluation compared the three algorithms: original SETO, Fast SETO and optimal assignment, in regard to the sequencer position in the original SETO algorithm. This “position” refers to the identifier of the node that performs the sequencer



(a) Original SETO (b) Fast SETO

Figure 3.4: Relative rates.

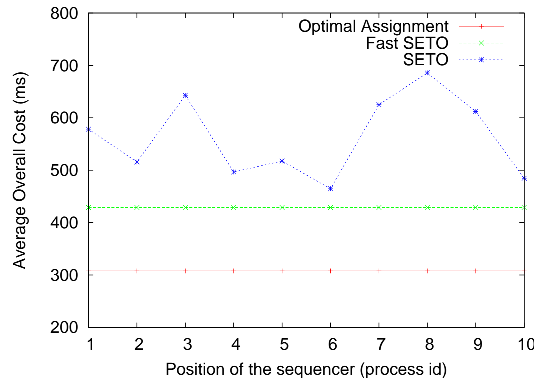


Figure 3.5: Sequencer's position.

role. For the experiments we used a network of 10 nodes with transmission rates uniformly distributed by all nodes and varying from 0 to 100 messages per second. For each sequencer position 20 network topologies where generated. The same 20 network topologies where used in all tests for the different positions, and the average overall cost of the 20 observations was used to produce the results presented in Figure 3.5.

The lines that represent the optimal assignment and the Fast SETO algorithm are obviously straight lines, because both algorithms results are not dependent of the sequencer position. As for the original SETO line, it is quite irregular and varies from little below $500ms$ to almost $700ms$, and always stays above the other algorithms lines. This clearly shows how the results of this algorithm depend on the sequencer position for a given network topology.

For this, the first step of the heuristic we propose for the Fast SETO algorithm may be quite useful for improving the results of the original SETO algorithm by helping to choose the best sequencer location.

Chapter 4

Improving the Appia toolkit

This section describes the work done in the Appia toolkit to fit the GORDA requirements. This includes a primary view protocol built on state of the art work, and several issues about security and management.

4.1 Providing primary partitions

One important property that should be provided by group communication is the notion of a primary partition. The composition of the group of replicas is dynamic. Sometimes a replica need to be excluded due to failures and need to be integrated after repairs. A group membership service tracks these changes and transforms them into views that are agreed upon as defining the group's current composition. Partitions in the group of replicas can happen due to failures in the cluster (network, switching hardware, among others). In asynchronous systems multiple views can also be the result of virtual partitions indistinguishable from real ones. This kind of failures generates partitions in the group of replicas.

A partitionable group membership service allows multiple views of the group, each corresponding to a different partition, to co-exist and evolve concurrently [1, 9]. In the context of database replication, this cannot happen if there are several replicas receiving and processing requests from the clients. A partition in the group membership can easily lead to a phenomenon usually called *split-brain*: the state in different replicas diverges and is not consistent any more. In contrast, a primary-partition group membership service maintains a single agreed view of the group at any given time. To achieve this requirement in a partitionable system, a primary-partition group membership service has to limit group membership changes to the primary partition and block all processes in non primary partitions by not delivering them any views [21] or by pretending that they have crashed so as to force rejoins after recovery.

4.1.1 Protocol overview

To provide this property in the group communication toolkit used in the project[1], we have enriched it with a protocol that provides primary partitions. This protocol is based on related literature [12, 2] and works as follows. Primary partitions are defined by majority quorums.

To bootstrap the system, the primary partition is defined at configuration time by assigning one element to be the primary. When the group membership changes due to joining, leaving or failures, the new primary partition is recalculated on all members and will be the one that contains a majority of processes from the previous primary partition. This is deterministic and ensures that only one partition exists at a time. Using this mechanism, a replica that belong to a primary partition can move to a non-primary partition when a view changes. In this case, the replication protocol only gets notified that the group has blocked and do not receive any view while it is not reintegrated in a primary partition.

4.1.2 Allowing recovery

It can happen that the replica never joins again correctly in a primary partition, because of intermediate views that could occur. These intermediate views could lead to other primary partitions that provide the opportunity to make progress in the system state. In this case, the replicas that lost this progress are forced to definitely leave the group. With this notification, a replica can reinitiate a new join process that includes a recovery process. The recovery process allows that the state of the replica can be updated with the missing state. This is possible because the group communication avoids that the state of the replicas diverges.

4.2 Management interfaces

In the scope of the project, the toolkit was improved with management interfaces that should be used to monitor the group communication system and dynamically change variables that will improve the system performance. The system performance can be influenced by several protocols that compose the group communication. Appia exports these features using the standard Java Management Extensions (JMX) programming interface.

Several protocols can be changed by an external process using the management interfaces. One of this protocols is the failure detection protocol that uses timeouts to suspect other members. If these values are not correctly set, the toolkit can generate false suspicions and generate unnecessary view changes that make the system unstable. Another protocol that can be changed at runtime is the total order switching protocol, in cases of network traffic changes. The primary partition protocol need also to be managed by an external process in the case of network partitions. In some cases, the protocol is unable to determine the primary partition and it blocks the system to avoid that the overall state diverges in different replicas. The management interface is used

[1] According to the Deliverable D1.1, the group communication toolkit mainly used in the project is Appia.

to define the processes that will continue to work (defining the next primary partition) in the recovery process.

4.3 Notification of services ensured

Some protocols needed to be changed to fit the requirements imposed by the group communication interface of the project[2], regarding optimistic deliveries and service notifications in general. The protocols that ensure total ordering of messages only delivered the messages to the application when the ordering was guaranteed. To use the optimistic assumptions, the protocols needed to be changed in order to make earlier deliveries and later deliver some notification that defines that some guarantee is ensured. This is mapped in the group communication interface using Services.

4.4 Communication security

The composition model adopted by the Appia group communication toolkit allows that each protocol can be constructed independently and, if it respects the Appia kernel interface, some protocol can be changed by another one that produces the same guarantee. The class of protocols that Appia already has is the one that provides the functionality of converting the events produced by the protocols to socket messages, and make the opposite conversion when a message arrives from a socket. This allows that the other protocols do not need to know about low level issues such as sockets and byte arrays. This also allows that protocol stack can be configured to use several solutions of the network.

One of the solutions that was provided in the scope of GORDA is an Appia protocol that uses Secure Socket Layer (SSL) instead of normal TCP or UDP sockets. This means that the communication can be secure if the group communication toolkit is configured to use this protocol. The protocol must be configured with the proper keys and certificates using a certificate management utility. The protocol supports any algorithm supplied by any of the registered cryptographic service providers, Such as the Digital Signature Algorithm (DSA) or the Rivest-Shamir-Adleman Encryption Algorithm (RSA).

[2]The interface was defined in the WP2 and is described in the Deliverable D2.3 and in [4]. It is also available in `http://jgcs.sf.net`.

Bibliography

[1] O. Babaoglu, R. Davoli, and A. Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. *Operating Systems Review*, 31(2):11–22, 1997.

[2] A. Bartoli and O. Babaoglu. Selecting a "primary partition" in partitionable asynchronous distributed systems. In *Symposium on Reliable Distributed Systems*, pages 138–145, 1997.

[3] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.

[4] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, October 2006.

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

[6] ILOG CPLEX. http://www.ilog.com/products/cplex/.

[7] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[8] 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. IEEE, 1993.

[9] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, page 343, New York, NY, USA, 1996. ACM Press.

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

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

[12] I. Keidar and D. Dolev. *Dependable Network Computing*, chapter Totally ordered broadcast in the face of network partitions, pages 134–143. Kluwer Academic, 2000.

[13] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *Proc. of the 9th Int. Symp. in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 346, Washington, DC, USA, 2001. IEEE CS.

[15] D. Nicol, J. Liu, M. Liljenstam, and G. Yan. Simulation of large-scale networks using SSF. In *Proceedings of the 2003 Winter Simulation Conference*, 2003.

[16] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proc. of the 12th International Symposium on Distributed Computing*, 1998.

[17] 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.

[18] D. Powell, editor. *Communications of the ACM*, chapter Special Issue on Group Communication, pages 50–97. Number 4 in 39. ACM, 1996.

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

[20] O. Rutti, P. Wojciechowski, and A. Schiper. Structural and algorithmic issues of dynamic protocol update. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, April 2006.

[21] A. Schiper and A. Sandoz. Primary partition "virtually-synchronous communication" harder than consensus. In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 39–52, London, UK, 1994. Springer-Verlag.

[22] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[23] António Sousa, José Pereiraa, Francisco Moura, and Rui Oliveira. Optimistic total order in wide area networks. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, pages 190–199. IEEE CS, October 2002.