Dynamic Total Order Protocol Replacement

Emili Miedes, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

{emiedes,fmunyoz}@iti.upv.es

Technical Report ITI-SIDI-2012/009

Dynamic Total Order Protocol Replacement

Emili Miedes, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

Technical Report ITI-SIDI-2012/009

e-mail: {emiedes,fmunyoz}@iti.upv.es

July 9, 2012

Abstract

The *dynamic total order protocol replacement* mechanism proposed in this paper allows an application to change, in run-time, the total order broadcast protocol it is using. In a *conventional* setting, without such a replacement mechanism, the application should be stopped, reconfigured and then restarted, in order to be adapted to a changing environment. The problem of this stop-and-restart approach is that it stops the normal operation of the application, reducing its availability. A dynamic replacement mechanism allows the application to switch from the current total order protocol it is using to a different one that performs better under the new operation conditions. Moreover, the switch should be performed in a transparent manner, from both the application and its users. This paper presents a software architecture for dynamically switching total order protocols, including the switching algorithm, its correctness justification and an experimental evaluation of its performance.

1 Introduction

Group Communication Systems are useful building blocks to develop highly available distributed applications. Such systems typically offer services like reliable multicast and broadcast message transports and group membership services. They usually offer additional services like total order broadcast, also known as *atomic broadcast*.

Such a service allows the nodes of a distributed application to broadcast messages while ensuring that all of the nodes receive the same sequence of messages, even if they get *disordered* by the underlying network mechanisms.

Total order has been studied for decades and a large number of theoretical [14, 10, 4, 11] and practical results [7, 6, 29, 12, 26, 3, 24, 1] have been shown. Nevertheless, all these *classic* results are *static* in the sense that the relationship between an application and the underlying group communication sytem is statically established (for instance, in compile time or, in the best cases, in configuration time). This means that if an application needs to change the *stack* of underlying protocols, a *reboot* of the communication layers is needed, thus causing an interruption in the application service and reducing its availability.

On the other hand, the results presented in some previous papers from this area [22, 21, 23] show that there is no single total order protocol that provides the best performance under any working conditions. In practice, this means that the election of a total order protocol may have a significant impact on the performance of the application. Specifically, the election of an *inappropriate* protocol may lead the application to get a worse performance. For this reason, the election of the protocol to use must be done carefully.

In this context, there are some problems that must be considered. First of all, it is not easy to *guess* the working conditions an application will have, unless it is a very specific application that has already been carefully evaluated. On the other hand, even when the working conditions of the application are

known beforehand, the election of the most suitable protocol requires from the designers of the application some knowledge about the available protocols. Moreover, it may happen that the working conditions of an application change during its execution, so the protocol first chosen as the most suitable becomes *unsuitable* due to these changes.

In practice, there should be some mechanism that allows the applications to use, in every moment, the most suitable total order protocol, according to different factors (application-dependent factors like the system load or message sending *patterns*, system-dependent factors like the underlying network and its topology, etc.). Moreover, such a mechanism should be *transparent* from the point of view of the protocols and the applications.

Such a mechanism offers several advantages. First of all, application designers do not need to *guess* the working conditions of the applications. They do neither need to know too many details about the protocols available nor about the best settings for each one of them. Moreover, such mechanism allows an application to *adapt to* changing working conditions and, in general, to get a better performance.

This paper studies the problem of dynamically changing the total order protocol used by an application. Section 2 presents a software architecture that allows such a dynamic replacement. Sections 3 and 4 present a ***Switching protocol*** that is able to perform the dynamic replacement. Section 5 discusses the properties of the ***Switching protocol*** and proves that it can act as a regular total order protocol. Section 6 presents an experimental work proving that besides being *effective*, the ***Switching protocol*** does not impose a significant performance overhead. Section 7 reviews some previous papers that tackle the dynamic switching topic. Finally, the paper is concluded in Section 8.

2 A Dynamic Protocol Replacement Architecture

An architecture for dynamically replacing Group Communication Protocols (GCPs) was presented in [23] and is explained in the sequel, discussing how it could be adapted to fit the needs presented above. Figure 1 shows a high level graphical description of the architecture.

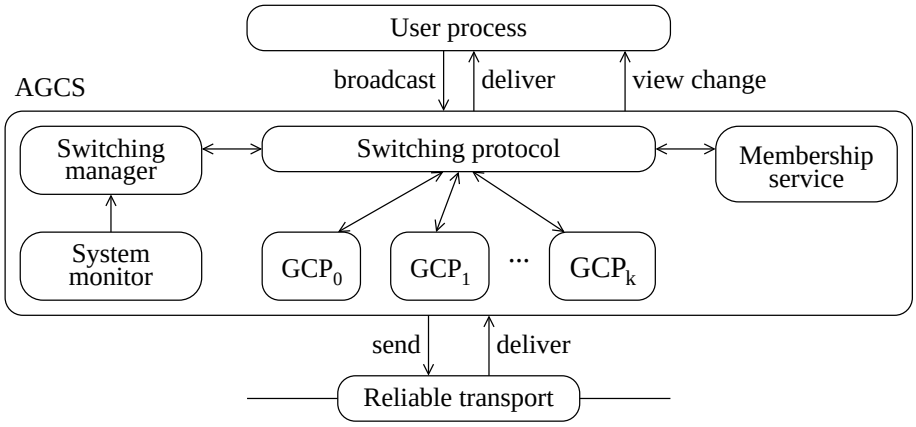

Figure 1: Architecture of a node

This architecture is based on a main component, called *Adaptive Group Communication System* (***AGCS***). As shown in the figure, the user process sits on top of the ***AGCS*** and this, in turn, relies on a regular reliable message transport layer. The ***AGCS*** wraps several standard group communication components (a number of GCPs, for instance, total order protocols and a membership service) and other specific components.

The ***Switching protocol*** implements the mechanism of replacing the GCPs at run-time. It captures the regular communication that occurs among the user process and the GCPs and performs the GCP replacement. The ***Switching manager*** is a component that decides when GCP changes should take place and which GCP should be installed. The ***Switching manager*** relies on a ***System monitor*** that keeps track of

several system and application measurable variables and parameters. The ***Switching manager*** collects the data provided by the ***System monitor*** and uses them to decide about GCP changes.

The architecture includes a ***Membership service*** component and a reliable transport layer. The ***Membership service*** provides notifications about changes on the set of nodes considered *alive* (due to joins of new nodes, node failures or node disconnections). Finally, the ***Reliable transport*** layer offers a regular *reliable* and *FIFO* message transport layer which ensures that a message sent to a destination is received by that destination unless it fails.

3 The Switching Protocol

This section provides an overview of the ***Switching protocol*** and then presents some notation details and its pseudocode algorithm.

3.1 Overview

During normal operation, when no GCP replacement is being carried on, the ***Switching protocol*** takes charge of the messages sent by the user process, which are forwarded to the current GCP. Incoming messages are received by the current GCP and directly handled by its protocol, which delivers them to the user process. The core of the ***Switching protocol*** does not take part in this process.

A GCP replacement starts when the ***Switching manager*** instructs the ***Switching protocol*** in a particular node to start a GCP change. The ***Switching protocol*** in this *initiator* node *to–bcasts* a PREPARE message to inform all the nodes about the new change. At every node, the ***Switching protocol*** stops relaying messages with the current GCP, instances and initializes a new GCP and starts relaying messages with it. Moreover, each node *to–bcasts* a PREPARE_ACK message to tell all nodes about the number of messages it has sent with the previous GCP and waits for a PREPARE_ACK from all the nodes.

In the meantime, the ***Switching protocol*** goes on receiving messages delivered to it by the previous GCP and forwarding them to the user application. The ***Switching protocol*** may also receive messages broadcast by the new protocol, as it has already been started in all nodes. These messages are not delivered to the user process yet, but stored in a local queue, until all messages broadcast with the previous GCP are delivered to the user process.

When the ***Switching protocol*** receives all the PREPARE_ACK messages it knows how many messages were sent with the previous GCP by each node. When all of them are finally received, the ***Switching protocol*** can finally discard the previous GCP. Then, it delivers to the user application the messages broadcast with the new GCP, which were locally queued. When all of them are delivered, the ***Switching protocol*** can go on using the new protocol as the only available one.

The protocol receives view changes from an independent membership service, for instance, when a node failure happens. If such a notification is received during a protocol change, the protocol basically *stops waiting for* messages from the failed node, so the protocol change can proceed when a node failure happens. An additional discussion is provided in Section 4.3.

Moreover, the protocol is able to manage consecutive protocol change requests. The protocol ensures that if a protocol change request is received by a node while a previous request is being handled, the current protocol change is completed and the next one is then handled. Additional details are given in Section 4.2.

3.2 Pseudocode

The pseudocode of the protocol is shown in Algorithms 1 and 2.

The protocol uses several *global variables*. A count of the GCP changes is maintained in variable k. It is initialized to 0 and incremented when a new GCP change is started. $Changing\_gcp$ is a flag to know if there is a GCP change in progress or it has already finished. $Live\_nodes$ is the set of live nodes as notified by the membership service.

The algorithm also uses a struct of type P for each GCP it manages. Thus, P_0 would be the struct for the first GCP used, P_1 would be the one for the second, etc. Such a struct contains several fields to store some state related to a GCP. Given a struct P_k, the expression $P_k.GCP$ is used to reference that GCP. The

Algorithm 1 The ***Switching protocol*** pseudocode (part I)

```
1:  INIT(G):
2:      current_k ← 0
3:      next_k ← 0
4:      changing_gcp ← false
5:      instance, prepare and initialize G
6:      call CREATE_P(P_{next_k}, G)
7:
8:  CREATE_P(p, g):
9:      p.GCP ← g
10:     p.k ← next_k
11:     p.sent ← 0
12:     p.other_sent[q] ← 0, for each process q in live_nodes
13:     p.ack_received[q] ← false, for each process q in live_nodes
14:     p.delivered[q] ← 0, for each process q in live_nodes
15:     p.deliverable ← {}
16:
17: TO-BCAST(m):
18:     if changing_gcp == true then
19:         to-bcast m with P_{next_k}.GCP
20:         P_{next_k}.sent++
21:     else
22:         to-bcast m with P_{current_k}.GCP
23:         P_{current_k}.sent++
24:     end if
25:
26: HANDLE_USER_MSG(m):
27:     if m.k == current_k then
28:         deliver m to the local process
29:         P_{current_k}.delivered[m.sender]++
30:         if changing_gcp == true then
31:             call FINISH_PENDING()
32:         end if
33:     else
34:         queue m in P_{m.k}.deliverable
35:     end if
36:
37: START(G'):
38:     to-bcast PREPARE(G') with P_{current_k}.GCP
39:
40: HANDLE_PREPARE(G'):
41:     next_k++
42:     changing_gcp ← true
43:     instance, prepare and initialize G'
44:     call CREATE_P(P_{next_k}, G')
45:     bcast PREPARE_ACK(current_k, P_{current_k}.sent) with P_{current_k}.GCP
46:
47: HANDLE_PREPARE_ACK(k, sent) from process q:
48:     P_k.other_sent[q] ← sent
49:     P_k.ack_received[q] ← true
50:     call FINISH_PENDING()
51:
```

Algorithm 2 The ***Switching protocol*** pseudocode (part II)

```
52: FINISH_PENDING():
53:     changing_gcp_aux ← false
54:     for j = current_k to next_k do
55:         if DELIVERY_FINISHED(j) then
56:             call END(j)
57:             current_k++
58:         else
59:             changing_gcp_aux ← true
60:             break
61:         end if
62:     end for
63:     changing_gcp ← changing_gcp_aux
64:
65: END(j):
66:     for all m in P_{j+1}.deliverable do
67:         if m is a user message then
68:             call HANDLE_USER_MSG(m)
69:         else if m is a PREPARE message then
70:             call HANDLE_PREPARE(m)
71:         else
72:             call HANDLE_PREPARE_ACK(m)
73:         end if
74:         remove m from P_{j+1}.deliverable
75:     end for
76:     destroy P_j.GCP
77:
78: HANDLE_VIEW_CHANGE(failed_nodes):
79:     remove failed_nodes from live_nodes
80:     call FINISH_PENDING
81:
82: DELIVERY_FINISHED(j):
83:     totalOtherSent ← 0
84:     totalDelivered ← 0
85:     for all q in live_nodes do
86:         if P_j.ack_received[q] == false then
87:             return false
88:         end if
89:         totalOtherSent+ = P_j.other_sent[q]
90:         totalDelivered+ = P_j.delivered[q]
91:     end for
92:     if totalOtherSent == totalDelivered then
93:         return true
94:     else
95:         return false
96:     end if
97:
```

Algorithm 3 The ***Switching protocol*** pseudocode (part III)

```
98:  INIT(G, sending):
99:      ...
100:     provide_sending_view ← sending
101:     changing_view ← false
102:
103: TO-BCAST(m):
104:     if changing_view == true and provide_sending_view == true then
105:         block call
106:     end if
107:     if changing_gcp == true then
108:         to-bcast m with P_{next_k}.GCP
109:         P_{next_k}.sent++
110:     else
111:         to-bcast m with P_{current_k}.GCP
112:         P_{current_k}.sent++
113:     end if
114:
115: HANDLE_VIEW_CHANGE(new_nodes, failed_nodes):
116:     changing_view ← true
117:     remove failed_nodes from live_nodes
118:     to-bcast NEW_VIEW(new_nodes, failed_nodes) with P_{next_k}.GCP
119:     call FINISH_PENDING()
120:
121: HANDLE_NEW_VIEW(new_nodes, failed_nodes):
122:     add new_nodes to live_nodes
123:     for all q in new_nodes do
124:         for j = current_k to next_k do
125:             P_j.other_sent[q] ← 0
126:             P_j.ack_received[q] ← false
127:             P_j.delivered[q] ← 0
128:         end for
129:     end for
130:     deliver (new_nodes, failed_nodes) to the local process
131:     if provide_sending_view == true then
132:         unblock call to TO-BCAST (if any)
133:     end if
134:     changing_view ← false
135:
```

$P_k.k$ field is the number of the replacement by which the $P_k.GCP$ is installed. In general, $P_k.k = k$. $P_k.sent$ is the number of user messages that have been broadcast by $P_k.GCP$. $P_k.other\_sent$ is an array that stores the number of messages sent by all the processes in iteration $P_k.k$ by means of $P_k.GCP$. Each entry of the array is initialized to 0 and updated when a new protocol replacement is started, using the information received from each process. The number of messages sent by process q is $P_k.other\_sent[q]$ and it is initialized to 0. $P_k.delivered$ is an array that stores the number of messages sent by all the processes delivered by the local process by means of $P_k.GCP$. $P_k.delivered[q]$ is the entry corresponding to the messages sent by process q. Each entry of the array is initialized to 0 and updated by the local process each time it receives a message from $P_k.GCP$. $P_k.deliverable$ is a list of messages delivered to the protocol by $P_k.GCP$. If $P_k.GCP$ is not the current protocol but a later one, the messages delivered by it cannot be directly forwarded to the user process. Instead, they are stored in $P_k.deliverable$, until all the messages sent with all the previous GCPs are delivered.

It is assumed that the managed GCPs provide a *to–bcast* primitive to broadcast (in total order) a message to all the nodes in the system. Given a message m, $m.sender$ denotes its sender.

The algorithm is composed by a set of *handlers* and *functions* which are executed as a response to external messages (sent by other nodes) and events (e.g. view change events produced by the ***Membership service***) or called from other event handlers and functions. These handlers and functions are *atomic*, i.e. two handlers or functions can not be executed concurrently.

The `INIT` function is executed only once, when the whole system is started. The `TO-BCAST` handler is invoked by the user application in order to broadcast a message (in total order). The `HANDLER_USER_MSG` handler is invoked by the GCPs to deliver incoming totally ordered messages to the ***Switching protocol***. The `START` function is executed when the ***Switching manager*** decides to start a new protocol change. The `HANDLE_PREPARE` and `HANDLE_PREPARE_ACK` are invoked by the GCPs to deliver `PREPARE` or `PREPARE_ACK` messages, to the ***Switching protocol***. The `FINISH_PENDING` function is invoked to try to finish as much pending protocol changes as possible. The `END` function is executed to finish a

protocol change. The HANDLE_VIEW_CHANGE handler is invoked by the membership service to deliver notifications on the membership view. The DELIVERY_FINISHED function is invoked to decide if all the pending messages needed to perform a protocol change have already been received.

4 Discussion

This section presents some issues that were not covered in Section 3 to simplify the presentation of the protocol. These issues cover the normal operation of the protocol and also its behavior in presence of failures.

4.1 Normal Operation

The ***Switching protocol*** offers a number of advantages over the protocols reviewed in Section 7. First of all, it does not block the sending of user messages. When a node is instructed to start a protocol switch, the sending of messages with the current GCP is disabled but message sending is immediately enabled with the new GCP. Moreover, it allows both protocols to coexist and work (i.e. to order messages) in parallel during the protocol change, until the old protocol is no longer needed. An important consequence is that the normal flow of messages is not delayed by slower processes. Even more, the delivery of messages to the user process is neither blocked. Indeed, when the old protocol is finally discarded and uninstalled, the ***Switching protocol*** immediately delivers to the user process the queued messages managed by the new GCP. After this step, regular delivery with the new protocol is enabled, thus keeping a *normal flow* of messages delivered to the user process.

On the other hand, for this mechanism to properly work, some issues must be considered. Firstly, it is needed some way to distinguish the messages broadcast with each GCP. A first solution consists in adding some *header data* in the regular messages but this solution would imply the need of knowing some implementation details, thus making the ***Switching protocol*** dependent on specific GCP implementations. A second option, general enough to fulfill this requirement is to encapsulate the regular user messages in other messages whose format is only known by the ***Switching protocol***. The protocol can include in these messages additional headers with all the needed meta-data. One of these headers stores the identifier of the GCP used to broadcast the encapsulated user message. From the point of view of the GCPs managed by the ***Switching protocol***, these protocol-dependent messages are as opaque as the regular user messages.

4.2 Concurrent Starts

A second issue is the ability of the ***Switching protocol*** to face concurrent starts of the switching procedure. Indeed, when several protocol switches are started concurrently by different nodes, the use of a total order broadcast protocol to broadcast the PREPARE messages forces that all the nodes receive those PREPARE messages in the same order.

Thus, multiple PREPARE messages can be received by a node. When a PREPARE message is received by a node, it starts a new $next\_k$ iteration, by creating a new P_{next_k} structure. The protocol starts sending messages with the new GCP and queueing in $P_{next\_k}.deliverable$ the messages delivered by it. Each time a new PREPARE message is received, a new iteration is started, even if there are some previous GCPs receiving messages.

When the current GCP delivers a message to the ***Switching protocol*** it checks if that message delivery allows to finish the execution of one or more iterations. For this, the FINISH_PENDING function is invoked. The only issue to worry about is the proper finalization of the iterations, in the same order they were started. This function checks that, for each iteration started, a corresponding PREPARE_ACK message has already been received from all the live processes and all the messages sent by them with the corresponding GCP have also already been received. In this case, the iteration can be considered finished, and the following iteration can be checked.

4.3 View Management

When no node failure happens, the behavior of the protocol is that shown in Algorithms 1 and 2. Nevertheless, the ***Switching protocol*** is able to react to failure notifications provided by an independent membership manager. These are received in the `HANDLE_VIEW_CHANGE` handler. This handler updates the local copy of the set of nodes considered alive and calls the `FINISH_PENDING` function. This function removes the failed nodes from the set that was checked in each pending iteration.

The reaction to view changes in these algorithms is actually minimum. Algorithm 3 extends the initial pseudocode shown in Algorithms 1 and 2. These extensions allow the protocol to provide view change notifications to the upper user process and also manage the join of new nodes. Regarding the first issue, two different alternative guarantees can be provided: *Same View Delivery* and *Sending View Delivery* [10].

If the *Sending View Delivery* property has to be guaranteed, the ***Switching protocol*** has to ensure that all the messages broadcast by the user processes are delivered to them in the view they were sent. In particular, the protocol has to ensure that all the messages broadcast with any of the *pending* GCPs are delivered *before* delivering the following view change notification to the user process. Moreover, once the ***Switching protocol*** learns about a node failure, it has to prevent the user process from sending more messages until the corresponding view change is delivered to it.

To this end, when the ***Switching protocol*** is informed about a node failure, it first blocks the sending of user messages. Then, it broadcasts a special `NEW_VIEW` message, with the last GCP started ($P_{next\_k}.GCP$). This message is broadcast with the last GCP started because it is not guaranteed that the previous GCPs are still available in all nodes. The `NEW_VIEW` message includes the set of nodes that compose the new view. After delivering all the pending user messages (those broadcast with any of the started GCPs, including the current one), this `NEW_VIEW` message is eventually delivered to the ***Switching protocol***. The ***Switching protocol*** can then forward the `NEW_VIEW` message to the user process, in order to notify the new view. Finally, it unblocks the sending of user messages.

A few assumptions must be made for this procedure to be correct. First of all, if the *Sending View Delivery* property has to be provided by the ***AGCS***, it must be ensured by the wrapped GCPs. This means that the GCPs must have some *flush mechanism* that ensures that when a node fails, before installing the new view, the pending messages are *flushed*. This mechanism may resend and forward messages, so all the alive nodes receive and deliver the pending messages, although the ***Switching protocol*** does not need to be aware of the *flush* procedure. The second assumption is that the `NEW_VIEW` messages broadcast by the ***Switching protocol*** by means of the GCPs, to inform about the view changes are not considered as regular application-level messages by the GCPs but interpreted as membership messages. When the ***Switching protocol*** sends a `NEW_VIEW` message through a GCP, informing about a view change, and especially about node failures, the GCP may start its *flush* protocol. This ensures that the pending user messages are finally delivered.

If the *Sending View Delivery* property is not needed, then the sending of user messages does not need to be blocked. The procedure to follow is thus the same than in the previous case except that the sending of user messages is not blocked. In this case, the user process can go on broadcasting messages after the ***Switching protocol*** receives the node failure notification. Nevertheless, these messages may be delivered to the user process (once totally ordered) after the ***Switching protocol*** delivers the view change to the user process, i.e., in a different view from the one they were sent in, although the total order property provided by all the GCPs ensures that, at least, each message is delivered in the same view to all the user processes. This way, the *Same View Delivery* property is ensured.

The ***Switching protocol*** is also able to manage the join of new nodes. Joins are notified as view changes. In fact, a view change can be viewed as a set of new nodes (nodes that join the system) and a set of nodes that fail.

In order to implement these features, Algorithm 3 uses two new global variables. The $provide\_sending\_view$ variable is a flag that sets whether the *Sending View Delivery* property has to be ensured. Its value is set to the value of the $sending$ parameter of the `INIT` handler. If it is set to $false$, then the *Same View Delivery* property is offered instead. Moreover, there is a $changing\_view$ global flag that maintains whether a view change is in progress.

The `TO-BCAST` handler is also modified. As a first action, it checks if a view change has been started and if the *Sending View Delivery* property has to be ensured. In this case, the user call to the `TO-BCAST`

is blocked. The rest of the handler is the same that the one shown in Algorithm 2.

The HANDLE_VIEW_CHANGE handler is also modified. First of all, a new parameter is added, to receive a set of new nodes (i.e., nodes that *join* the system). Then, it broadcasts a special NEW_VIEW message, by means of the last GCP started. Finally, the FINISH_PENDING function is invoked, as in Algorithm 2.

The NEW_VIEW message is received in the new HANDLE_NEW_VIEW handler. First, the new nodes are added to the local copy of the set of nodes considered alive. The P data structures from $P_{current\_k}$ to P_{next_k} are updated, to initialize the state corresponding to the new nodes. Then the view change is delivered up to the user process. Finally, in case the *Sending View Delivery* property was required, it unblocks the execution of the TO-BCAST handler.

Another issue related to the notification of node failures must be addressed. When a node fails it may happen that, in several nodes, the corresponding membership service notifies to the ***Switching protocol***, which would broadcast its NEW_VIEW message. The result is a number of NEW_VIEW messages representing the same node failure are broadcast and received by all nodes. To avoid the multiple notification of a view change to the user processes a simple solution can be adopted.

The ***Switching protocol*** keeps a *view counter* as a global variable. It is initialized to 0 and incremented each time a NEW_VIEW is delivered to the ***Switching protocol*** and then forwarded to the user process. Each NEW_VIEW message is tagged with the current value of the counter when it is broadcast. If the ***Switching protocol*** receives different NEW_VIEW messages with the same value of the *view counter*, it considers the first one and then discards the rest. As the NEW_VIEW messages are broadcast in total order, using the last GCP started, all nodes keep the same NEW_VIEW message and discard the same other messages.

5 Properties of the *Switching protocol*

This section presents some properties of the ***Switching protocol*** and some reasoning about their correctness. First, some lemmas are proposed.

Lemma 1: Downwards Validity *If a user process in a correct node broadcasts a message m, then exactly one of the GCPs of that node eventually broadcasts m exactly once.*

Proof. In the TO-BCAST handler, each message sent by the user process is immediately broadcast exactly once, by any of the GCPs currently managed by the ***Switching protocol*** (lines 18–24).

Considering the modifications presented in Algorithm 3, in case the *Sending View Delivery* property is requested and a view change happens, the following message broadcast by the user process may be blocked. In this case, it should be shown that the sending is not blocked infinitely.

First, when a view change is notified, then a NEW_VIEW message is broadcast (line 118). By the *Validity* property of the GCP used to broadcast the NEW_VIEW message, this is eventually delivered by the local node and handled in the HANDLE_NEW_VIEW handler. In this handler, the user process is finally unblocked (line 132) and the message can finally be broadcast, exactly once and using exactly one GCP (lines 107–113). □

Lemma 2: Upwards Validity *If a GCP delivers a message m to the **Switching protocol**, then the **Switching protocol** eventually delivers m to the user process.*

Proof. It has to be shown that the ***Switching protocol*** does not indefinitely retain a message delivered to it by a GCP. First, if a message m is delivered to the ***Switching protocol*** by $P_{current\_k}.GCP$, then it is immediately delivered to the user process (line 28). If the message is delivered by $P_{k'}.GCP$ (where $current\_k < k' \leq next\_k$), then it is stored in $P_{k'}.deliverable$. In this case, it has to be shown that the message is not retained in that queue infinitely. In other words, it has to be shown that all iterations of the protocol previous to k' are eventually finished.

If m was broadcast with $P_{k'}.GCP$ (with $current\_k < k'$), then we know that a finite number of messages were broadcast with $P_j.GCP$ ($\forall j : current\_k \leq j < k'$). By the *Validity* and *Uniform Agreement* properties of these GCPs, it is known that all those messages are eventually delivered to the ***Switching protocol*** and, by Lemma 1, eventually delivered to the user process. For the same reason, we also know that all

the corresponding PREPARE_ACK and PREPARE messages (used to finish an iteration and start the next one, respectively) are eventually delivered to the ***Switching protocol***. Then, all the iterations previous to $P_{k'}$ are eventually finished. An iteration P_j is finished when all the messages broadcast with the $P_j.GCP$ are delivered to the ***Switching protocol*** (as decided by the DELIVERY_FINISHED function). At the end of the iteration P_j, all the pending messages broadcast with $P_{j+1}.GCP$ (those stored in $P_{j+1}.deliverable$) are delivered to the user process (lines 66–75). Then, $current\_k$ is incremented (line 57). Eventually, $current\_k$ reaches k' and message m is finally delivered to the user process. □

Lemma 3: Local Integrity *The **Switching protocol** delivers a message m to the user process at most once, and only if m has been delivered to the **Switching protocol** by exactly one of the GCPs of the local node.*

Proof. First of all, the ***Switching protocol*** delivers the message to the user process at most once. If the message is delivered by the current GCP ($P_{current\_k}.GCP$) then, it is directly delivered (line 28). If the message is delivered by a later GCP ($P_{k'}.GCP$, with $current\_k < k'$), then it is first queued (in $P_{k'}.deliverable$). By Lemma 2, the message is eventually delivered to the user process, exactly once (lines 66–75).

On the other hand, it has to be proved that a single message cannot be delivered to the ***Switching protocol*** by more than one GCP. Let's suppose that a message is delivered to the ***Switching protocol*** by two different GCPs. The *Uniform Integrity* property offered by these GCPs ensures that they previously sent the message. Nevertheless, this is not possible since the ***Switching protocol*** sends each message only with one of the GCPs (lines 28 and 34). □

Lemma 4: Change Safety *The **Switching protocol** does not deliver to the user process a message m delivered to the protocol by $P_k.GCP$ after having delivered to the user process a message m' which was delivered to the protocol by $P_{k'}.GCP$, where $k < k'$.*

Proof. If no view change happens, the TO-BCAST handler broadcasts the user messages by means of $P_{current\_k}.GCP$ (line 22). As the ***Switching protocol*** does not keep a P_k previous to $P_{current\_k}$, then no message can be broadcast with a previous GCP.

If a GCP change happens, the TO-BCAST handler broadcasts the user messages by means of $P_{next\_k}.GCP$ (line 19). The value of $next\_k$ is incremented each time a GCP change is started (line 41), so $P_{next\_k}.GCP$ is always the last GCP that has been started. If a message is broadcast with $P_{next\_k}.GCP$, then any message subsequently broadcast will be sent with the same GCP or a later one. □

Property 1: Validity *If a process in a correct node broadcasts a message m, then the **Switching protocol** eventually delivers m to it.*

Proof. If no GCP change happens, message m is sent with the current GCP ($P_{current\_k}.GCP$). By its $Validity$ property, the GCP eventually delivers m to the ***Switching protocol*** (in the same node). According to Lemma 3 (*Local Integrity*) stated above, the ***Switching protocol*** eventually delivers the message to the user process.

If a GCP change happens, Lemmas 1 (*Downwards Validity*) and 2 (*Upwards Validity*) ensure that the ***Switching protocol*** does not indefinitely retain the *outgoing* messages sent to it by the user process nor the *up-going* messages delivered to it by the GCP. □

Property 2: Uniform Agreement *If the **Switching protocol** in a node, whether correct or faulty, delivers a message m to the user process, then the **Switching protocol** in all correct nodes eventually deliver m to their corresponding user processes.*

Proof. Let's suppose that, in one of the nodes, the ***Switching protocol*** delivers a message to the user process. By Lemma 3 (Local Integrity), the message must have been delivered to the ***Switching protocol*** by

one of the GCPs. By the *Uniform Agreement* property of the GCPs, in all the correct nodes, the GCP delivers the message to the ***Switching protocol*** and by Lemma 2 (Upwards Validity), the ***Switching protocol*** eventually delivers up the message to the user process in all correct nodes. □

Note that when the GCPs do not satisfy the *Uniform Agreement* property but just a *Non-uniform Agreement* property, then the property satisfied by the ***Switching protocol*** is not *Uniform Agreement* but just the corresponding *Non-uniform Agreement* property.

Property 3: Uniform Integrity *For any message m, the **Switching protocol** of every node, whether correct or faulty, delivers m at most once to the user process and only if m was previously broadcast by its sender.*

Proof. First of all, it has to be shown that a user process does not deliver a message twice.

First, by Lemma 3, the ***Switching protocol*** cannot deliver twice the same message.

Moreover, it has to be shown that the ***Switching protocol*** only delivers a message to the user process if the message was previously broadcast by its sender node.

First, it is known that the ***Switching protocol*** only delivers to the user process messages that have previously been delivered to it by one of the GCPs (lines 28). By the *Uniform Integrity* of the GCPs, this only happens after the GCP in the sender node has broadcast the message. The ***Switching protocol*** itself ensures that this can only happen after it has broadcast the message through the corresponding GCP in the sender node. □

Property 4: Uniform Total Order *If the **Switching protocol** in any nodes p and q, whether correct or faulty, both deliver messages m and m', then the **Switching protocol** in p delivers m to its user process before m' if and only if the **Switching protocol** in q delivers m to its user process before m'.*

Proof. Let's suppose that the ***Switching protocol*** in both nodes p and q delivers two messages m and m'. If p delivers both m and m' using the same GCP, by the *Uniform Total Order* property of the GCP and by protocol construction, it is known that all the nodes will deliver m and m' in the same order, using the same GCP.

Now let's suppose that p delivers m using $P_k.GCP$ and delivers m' using $P_{k'}.GCP$, with $k < k'$. Then, q also delivers m using $P_k.GCP$ and m' using $P_{k'}.GCP$. Moreover, by Lemma 4 (*Change Safety*), as m has been broadcast using $P_k.GCP$, q delivers m to the user process before delivering any other message broadcast by $P_{k'}.GCP$, which means that q delivers m prior to m'.

The reasoning is also valid if p or q fail after delivering m and m', respectively. On one hand, p and q deliver m and m', as long as $P_k.GCP$ and $P_{k'}.GCP$ satisfy the *Uniform Total Order* property. On the other hand, by Lemma 4 (*Change Safety*), both nodes deliver all the messages broadcast by $P_k.GCP$ before starting to deliver messages broadcast by $P_{k'}.GCP$. As a result, both p and q deliver m before delivering m'. □

6 Experimental Evaluation of the *Switching protocol*

This section presents an experimental evaluation of the ***Switching protocol***. First, the environment and the methodology used to perform the evaluation are described. Then, some results that show the effectiveness of the protocol are presented.

The overall architecture of the system running in each node is shown in Figure 2, which is a simplified version of that depicted in Figure 1. The application acts as a *client* of the ***Switching protocol*** which in turn wraps several of the total order protocols implemented to perform the experimental evaluations.

The message transport layer is a new transport layer implemented on top of the JBoss Netty 3.2.4 networking library [16]. Netty is a client/server library that implements the Java NIO specification [2] and offers asynchronous event-driven abstractions for using I/O resources.

On the other hand, the implemented system does not include a *Membership service* since view changes are not considered in this evaluation. Moreover, the implemented system does not include the *System monitor* or the *Switching manager* shown in the original figure. Instead, the test application itself is in

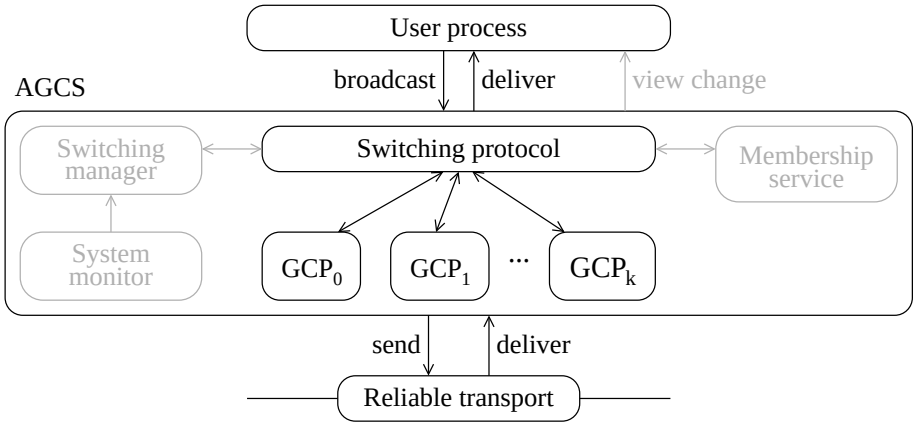

Figure 2: Architecture of a node (simplified version).

charge of issuing the *start-of-change* events to request a GCP change, to any of the available *pre-loaded* GCPs (***UB***, ***TR***, etc.), as described later.

The application is executed in a system composed of four nodes. Each node is a different physical machine with an Intel Pentium D 925 processor running at 3.0 GHz and 2 GB of RAM, running Debian GNU/Linux 4.0 and Sun JDK 1.5.0. The nodes are connected by means of a 24-port 100/1000 Mbps DLINK DGS-1224T switch. The switch keeps the nodes isolated from any other node, so no other network traffic can influence the results.

The test application is a regular Java console application that is run in each of the four nodes of the system. In each node, the application broadcasts a sequence of messages by handling them to the ***Switching protocol*** as if it were a regular total order protocol.

The messages are broadcast at a uniform sending rate, configured externally. No other message flow control mechanism has been used.

To perform the evaluation of the ***Switching protocol***, the test application has been run under different configurations. In a first set of executions, the ***Switching protocol*** is configured to use the ***UB*** (sequencer-based) and the ***TR*** (privilege-based) protocols. The application was configured to periodically request a GCP change each 5000 ms. Thus, the ***Switching protocol*** starts using the ***TR*** protocol and after 5000 ms the ***Switching protocol*** is asked to *switch* to ***UB***. After the next 5000 ms, the application asks to change to ***TR*** and so on.

In each test, the application was configured to broadcast messages at a fixed sending rate. Different tests have been run using rates of 40, 60, 80, 120 and 130 messages broadcast per second and node. Thus, the global sending rates range from 160 to 520 messages per second.

Each execution measures the *delivery time* of the messages, computed as the time observed by the application in a given node, from the moment in which it broadcasts the message to the moment in which it receives back the message, once totally ordered. This means that for each node, a series of delivery times is gotten corresponding to the series of messages broadcast by that node.

Moreover, in order to know the *distribution in time* of the message deliveries, the number of messages that are delivered in each *hundredth* of a second is considered. These numbers allow us to know if there is a regular *flow* of messages being delivered.

This set of experiments is repeated, using the ***UB_PRIO*** and ***TR_PRIO*** protocols.

Figure 3 shows the *delivery times* recorded by a single node in the *first* set of experiments, with ***TR*** and ***UB*** and a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node. Figure 4 shows the corresponding amount of messages delivered in each hundredth of a second, using a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node.

Figure 5 shows the *delivery times* recorded by a single node in the *second* set of experiments, with ***TR_PRIO*** and ***UB_PRIO*** and a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node. Figure 6 shows the corresponding amount of messages delivered in each hundredth of a second,

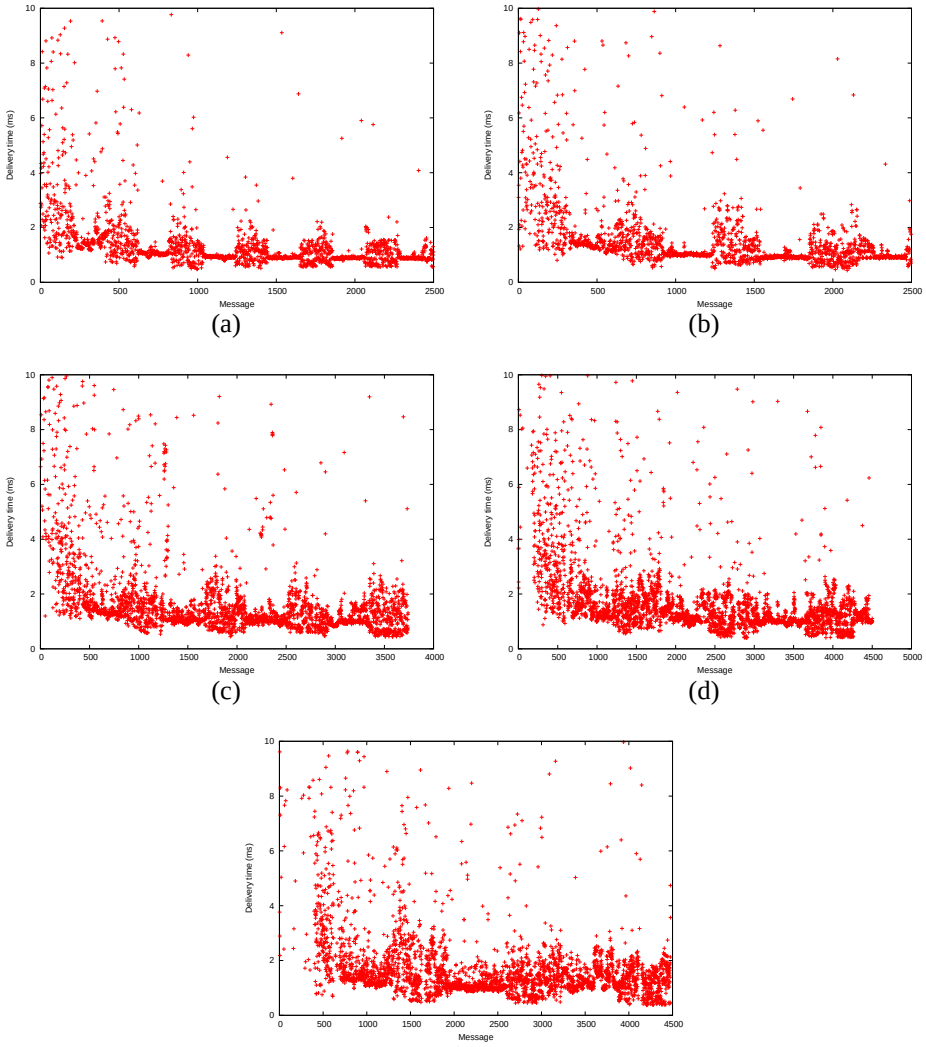

Figure 3: Delivery times with ***TR*** and ***UB***: (a) 40 msg/s, (b) 60 msg/s, (c) 80 msg/s, (d) 120 msg/s, (e) 130 msg/s.

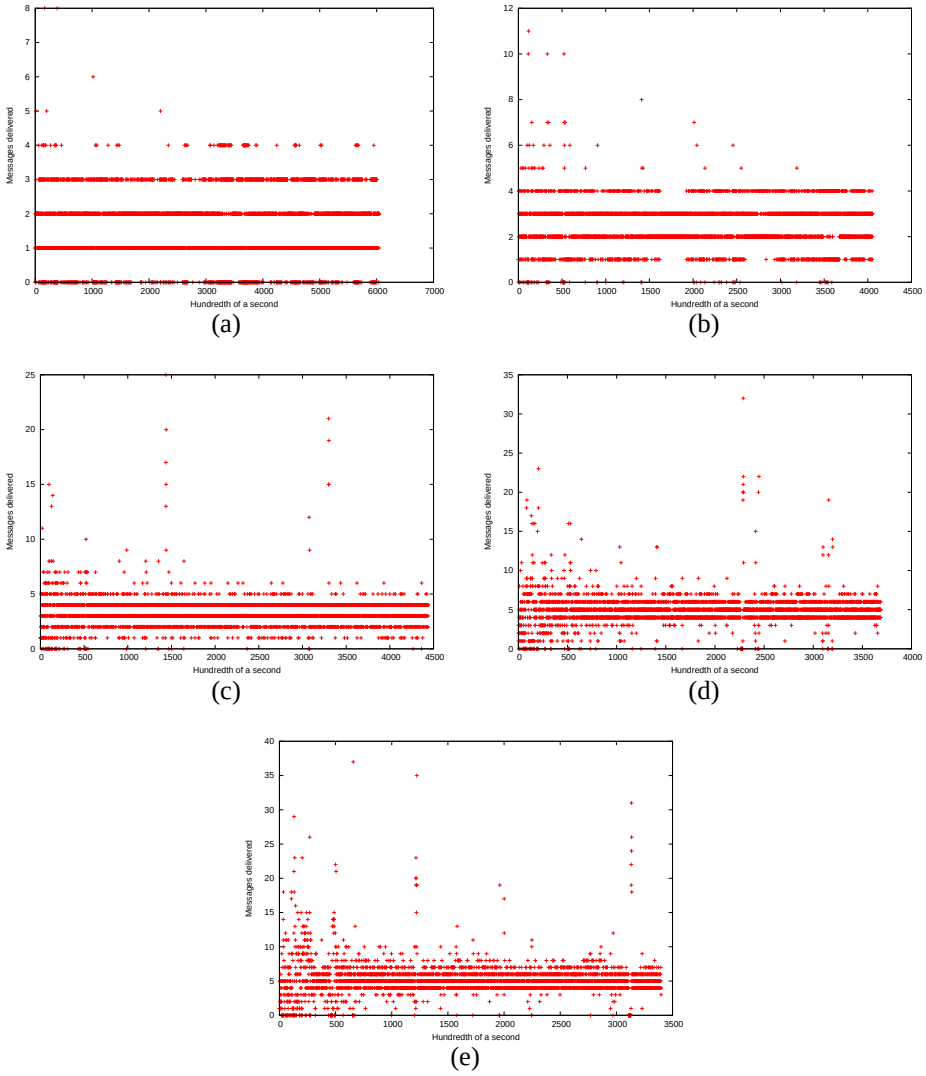

Figure 4: Messages delivered by hundredth with ***TR*** and ***UB***: (a) 40 msg/s, (b) 60 msg/s, (c) 80 msg/s, (d) 120 msg/s, (e) 130 msg/s.

using a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node.

Figure 3 shows the delivery time of the messages. Each test consists of a series of messages, which are delivered in total order. In a given series, the i_{th} message is represented by $x = i$ and its delivery time (in milliseconds) is $y(i)$. As explained above, in these tests the ***Switching protocol*** switches between two total order protocols (e.g. ***TR*** and ***UB***). This means that in any test, the application delivers a *sub-series* of the messages with the first protocol (in Figure 3.a, about 200 messages with ***TR***), then it delivers another *sub-series* with the second protocol (in Figure 3.a, another 200 messages, with ***UB***) and so on.

These figures show that the ***Switching protocol*** does not increase the delivery time of the messages.

These figures show two different *distributions* of the message delivery times. The delivery times of the messages delivered with ***TR*** show a higher *variability* than the times corresponding to messages delivered with ***UB***. The reason of this behavior is because according to the ***TR*** protocol, a node needs to have the *privilege* to send messages. In ***TR*** and ***TR_PRIO***, this is implemented by means of a rotating *token message*. To broadcast a (totally ordered) message, the nodes typically have to wait some *variable time*, until they get the token. This extra delay is the main responsible of the higher *variability* observed in Figures 3.a to 3.e.

As the sending rate is increased, the delivery times got with ***UB*** tend to increase because the sequencer node is more and more *busy* sequencing messages. The time increment becomes more and more variable, thus increasing the variability of the final message delivery times. Nevertheless, these differences are not actually so important, since even in these cases, the figures show that the ***Switching protocol*** is not introducing any significant delay in the message delivery times. This can be checked by analyzing the delivery times in each *sub-series*. In case that the ***Switching protocol*** introduced a delay in the delivery times, this delay would have been noticeable. Specifically, the delivery times at the beginning of each *sub-series* would have been noticeably higher than the delivery times of the rest of the *sub-series*. As the figures show, the delivery times in a given series are quite similar and comparable among them (apart from several punctual times that can be considered *anomalous*). From this behavior, it can be concluded that the ***Switching protocol*** is not introducing any significant delay in the message delivery times.

On the other hand, those figures assess the cost of delivering each message but they do not provide any information about how that message delivery is being *distributed* over time. Figure 4 shows the number of messages delivered every hundredth of a second (i. e. the interval [0:100] corresponds to one second). These graphs allow to know about the message delivery over time.

For instance, Figure 4.a shows that in most of the *one hundredth of a second* intervals, the ***Switching protocol*** is delivering between 1 and 3 messages. As the sending rate is increased (Figures 4.b to 4.e), this delivery rate also increases. For instance, in Figure 4.e, between 5 and 6 messages are delivered in each *one-hundredth* interval.

The importance of these figures is that in all cases, the number of messages delivered by hundredth of a second follow a *quite regular distribution*, in spite of the successive protocol changes that have happened. These figures also show a small number of anomalous values but it can be seen that they do not happen at instants of time which are multiple of 500 hundredths of second (5000 ms) but at any time, which means that they are not directly caused by a protocol switch, but by some other reason (for instance, due to the thread scheduling policies of the operating system or the Java Virtual Machine). Thus, it can asserted that the number of messages delivered per time unit does not depend on whether a protocol switching is being carried on and for this reason, it can be concluded that the ***Switching protocol*** is not producing interruptions or delays in the *flow* of message deliveries.

Figure 5 shows the corresponding results when using the ***TR_PRIO*** and ***UB_PRIO*** protocols. These results are very similar to those depicted in Figures 3 and 4, which allow to conclude that the ***Switching protocol*** is working properly when switches among any kind of total order protocols (prioritized or not): it does not impose significant time overheads in the message delivery times and it does not interrupt or delay the message delivery.

This shows that the ***Switching protocol*** is useful to adapt an application to changing requirements and load conditions. The figures presented show that some total order protocols are able to minimize the dispersion of the message delivery times while others lead to a reduction in the mean delivery time of the messages. The proposed switching support allows the applications to switch among different total order protocols under changing conditions, without suffering significant performance penalties during the protocol switch.

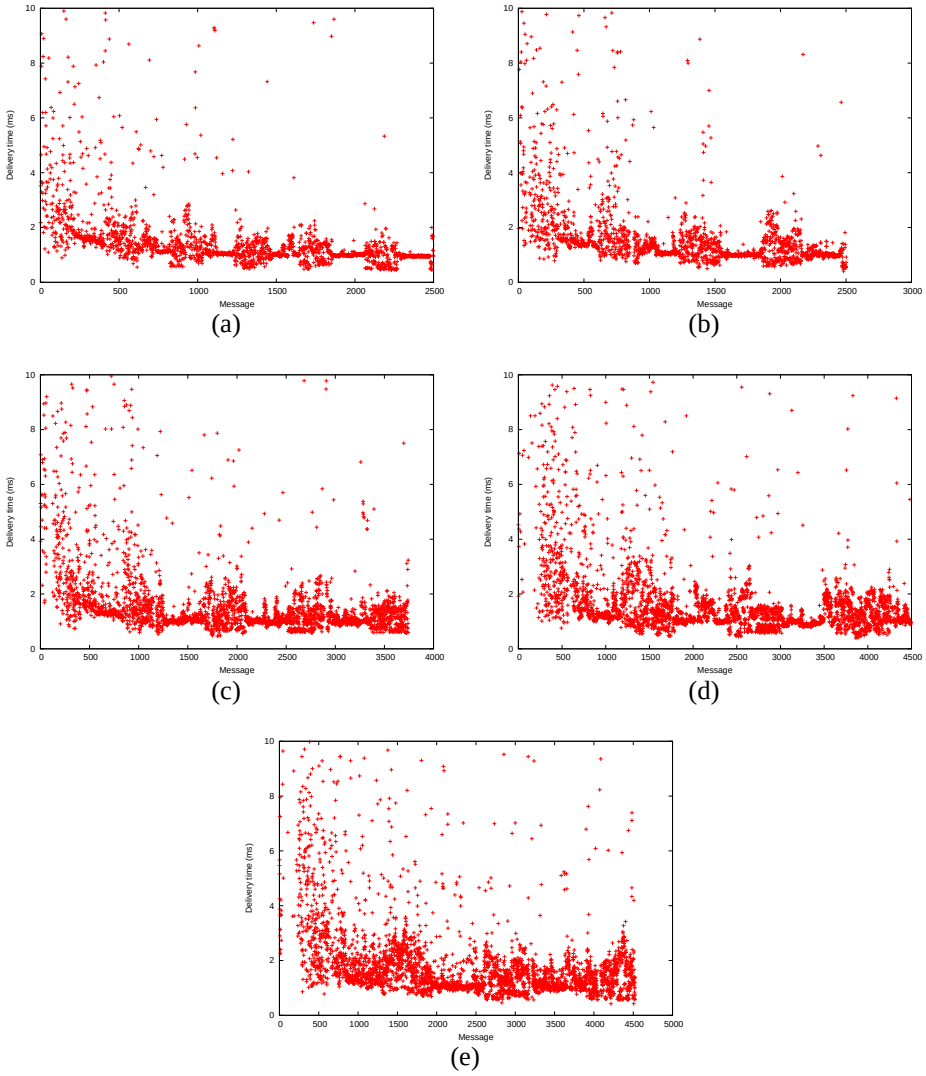

Figure 5: Delivery times with ***TR_PRIO*** and ***UB_PRIO***: (a) 40 msg/s, (b) 60 msg/s, (c) 80 msg/s, (d) 120 msg/s, (e) 130 msg/s.

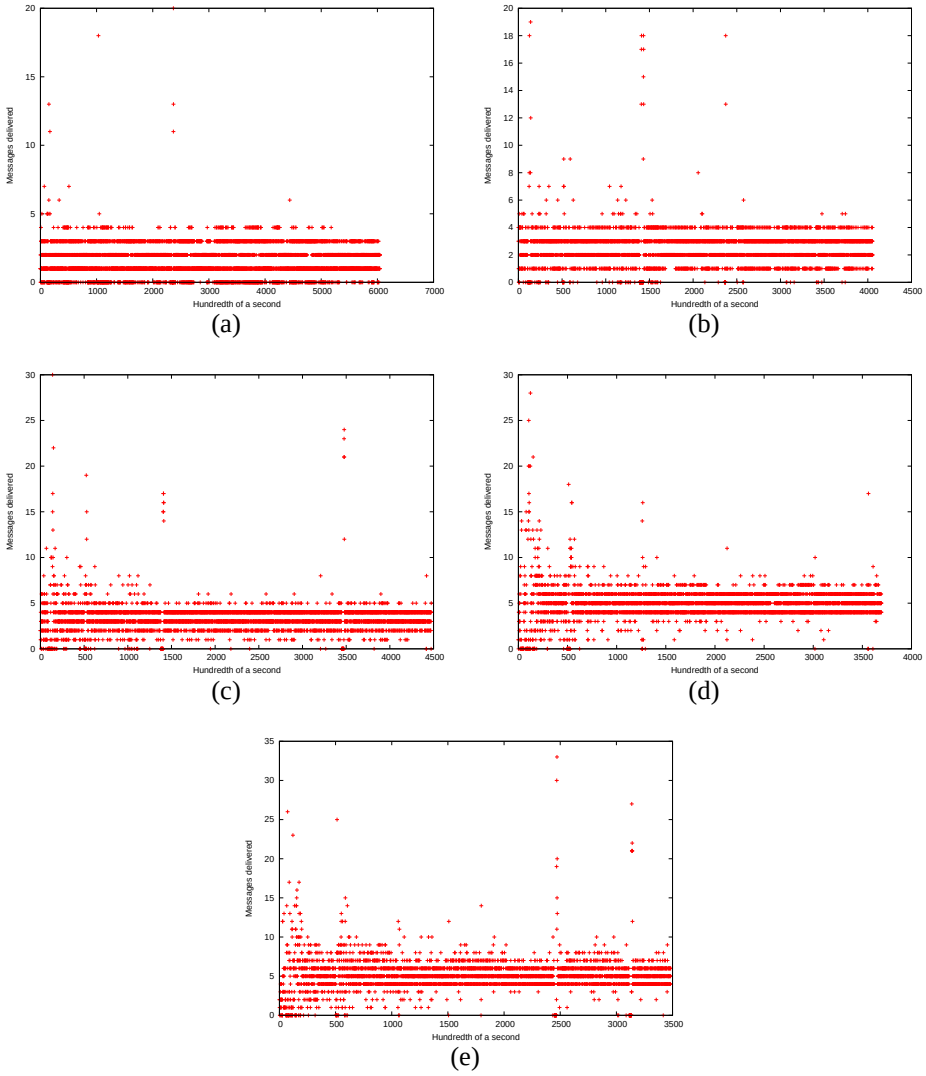

Figure 6: Messages delivered by hundredth with ***TR_PRIO*** and ***UB_PRIO***: (a) 40 msg/s, (b) 60 msg/s, (c) 80 msg/s, (d) 120 msg/s, (e) 130 msg/s.

7 Related Work

There have been other papers proposing some dynamic switching mechanism allowing the replacement of one or several services or modules by other that are better tailored to the current context. Such proposals are the key for obtaining an adaptive system and are described in the sequel, in chronological order.

The Ensemble system [15] is a group communication system based on the configuration and use of a *stack of protocols*. Each protocol of the stack provides a different service (message transport, group membership, ordering, etc.) to the application or to other protocols of the stack. In [28], a Protocol Switch Protocol (PSP) is proposed. The PSP is an Ensemble protocol that allows the dynamic replacement of the full protocol stack used by Ensemble. The PSP is a two-phase commit protocol (2PC) [13, 18]. The protocol includes some fault-tolerance support that tolerates the loss of messages (by means of retransmissions) and the node failures or disconnections. The PSP presents a significant disadvantage. As it is composed of two independent parts and the second part is not started until the first one is completed, the regular operation of the application is somehow blocked. The fact that the whole protocol stack is replaced is actually another inconvenience. Indeed, there is no way to replace a single protocol in a given protocol stack without having to stop and replace all the stack protocols.

In [19], the authors present an alternate mechanism to the switching protocols based on a 2PC technique. The idea is to make the switching more scalable, by avoiding the dependency on a single coordinator node and reduce the delay imposed by the transition from the older protocol to the new one. This alternative consists in defining *switching functions* that are used to switch from the state kept by a protocol to the state used by another protocol. In run-time, during a dynamic protocol switching, the use of such functions allow the nodes to go on working with the new protocol, which starts by managing the messages *inherited* from the first protocol and then goes on with the new messages.

In [20] a second *Switching Protocol* (SP) for the Ensemble system is presented. Unlike the protocol presented in [28], the SP allows the replacement of a single protocol of the Ensemble's protocol stack. The protocol is presented as a *wrapping* that sits on top of a number of alternative protocols that offer the same service, i.e. the same guarantees. This wrapping protocol offers those guarantees to the protocol layered about it, which, in fact, does not need to know about its *wrapping* nature. When it operates in *normal mode*, it just forwards up and down the messages sent by and delivered to its neighbor layers. When it operates in *switching mode*, it performs a protocol replacement. As in [28], the SP assumes some mechanism that decides about when the current protocol has to be changed. Thus, the protocol replacement starts when some *oracle* chooses a node as a replacement *manager*. The protocol operation is similar to that of [28] but there are some differences. First of all, the communication among the manager and the rest of nodes is no longer based on broadcasts. Instead, a logical ring is formed among all nodes and a token is forwarded from node to node along the ring. The token has a *mode* field that identifies the phase of the protocol. There are three protocol phases. This protocol has some drawbacks related to its *blocking* nature. First of all, it prevents nodes from sending messages with both the current and the new protocol until they are in the third token round. Moreover, the structure of the protocol, based on three rounds along the ring imposes a significant delay. Furthermore, this delay is increased by the blocking third round.

An adaptive architecture for run-time protocol switching is proposed in [9, 8], designed for Cactus [5], a framework for building distributed protocols and applications. A Cactus application is based on a stack of layered components, each offering a service. Some of these components may be *adaptive*, including different implementations of the same service. Initially, one of the available implementations of a given component is chosen. This architecture allows to change, in run-time, the current implementation of a service to one of the other available implementations of the service, in order to adapt to changing environments or contexts. For this, each component includes an *adaptor*, which is a module that collaborates with the service implementations to perform the replacement. The protocol change procedure is actually an abstract generic protocol, composed of three phases. A first phase is the detection of some changing environment or application parameters. A second phase, closely related to the first one, includes the election of the new implementation of the service. The third phase is the *adaptation* phase, which consists of three steps: a) preparation, b) *outgoing switchover* and c) *incoming switchover*. The preparation step includes all the actions needed to start and prepare the switching from one implementation of the service to the new one. It finishes with a *synchronization barrier*. Once all the participating nodes reach this barrier, they can proceed with the next step. The outgoing switchover is the step by which the flow of outgoing messages

that arrive to a service implementation are *redirected* to a different implementation of the service. The incoming switchover is a similar message *redirection*, applied to incoming messages.

Rütti et al. [27] consider the problem of *Dynamic Protocol Update* as a particular case of the more general *Dynamic Software Update* problem. The solution proposed is based on two *switching algorithms* that allow the dynamic replacement of one of the protocols in the stack used by an application. There is a switching protocol to replace the consensus protocol of the stack and another switching protocol to replace the atomic broadcast protocol. This solution is aimed at the SAMOA framework [30] but the basic idea may be applied to other protocol stack-oriented frameworks. According to the architecture proposed, one of the switching protocols is placed in the protocol stack, just above the protocol to change. When no protocol change is to be done, the switching protocol simply forwards up and down the messages sent by and delivered to the application. During a protocol change, the switching protocol intercepts the application messages. The general idea of *interception* includes delaying and resending messages. Both algorithms guarantee that the *service requests* performed with the current protocol (consensus or atomic broadcast) are finished before starting the operation with the new protocol. The operation of the atomic broadcast switching protocol actually relies on the atomic broadcast protocol to be replaced. When a node decides to start a protocol change, it broadcasts a special message with the current atomic broadcast protocol. When a node receives this special message, it performs the protocol replacement, by installing and activating the new protocol. If there are some pending messages sent with the old protocol they will be discarded by all nodes at delivery time and resent by their corresponding senders, using the new protocol. Some performance evaluation of both protocols is also presented. As shown in the graphical results, the need to resend some messages during the execution of the protocol change algorithm has a negative impact on the latency of a number of messages.

Mocito and Rodrigues [25] propose another switching protocol for total order protocols It avoids blocking message sending with the new protocol so the flow of application messages is never blocked. It sets a point in time from which no more messages are sent with the current total order protocol. Incoming messages broadcast with the new protocol are queued until all the pending messages are delivered with the current total order protocol and the protocol switching is completed. In order to deactivate the total order protocol being replaced, each node broadcasts an acknowledgement message as the last message broadcast using the current total order protocol. Upon reception of all such acknowledgement messages, a given node knows that no more messages will be sent with the current total protocol so the node can deactivate it.

Karmakar et al. [17] describe the use of a switching protocol to dynamically change the broadcast protocol used by a network of nodes. A broadcast protocol based on a Breadth-First Search tree yields lower message latencies when the network load is low. On the other hand, a broadcast protocol based on a Depth-First Search reduces the load on individual nodes when the global network load is higher. The mechanism discussed in [17] can switch between two broadcast protocols, one based on a BFS tree and another based on a DFS tree. The core of the mechanism is the construction of the spanning tree used by the broadcast protocol.

8 Conclusion

This paper reviews the problem of dynamically replacing the total order broadcast protocol used by a distributed application. It provides a new, non-blocking, highly concurrent switching protocol, fully integrable with existing independent membership services. Moreover, this protocol admits concurrent starts of the switching procedure.

The paper includes an extensive description of the switching protocol, a pseudocode algorithm and a discussion of the properties offered by the switching protocol that allow it to behave like a regular total order protocol. It presents an experimental evaluation of its operation.

Although this switching protocol was designed to allow the dynamic replacement of regular total order broadcast protocols, it can also be used to replace *prioritized* total order broadcast protocols, without any further modification.

To argue about this, it may be considered that the prioritized protocols presented in [22, 21] behave like regular total order protocols and that *Prioritization* is a property that can be observed on the sequence of messages they totally order. These protocols can be wrapped in an architecture like the one presented in

Figure 1. As long as the order of the sequence of messages provided by a given GCP is preserved by this architecture, the *Prioritization* property will be preserved. Moreover, as the switching protocol only relies in the regular properties offered by common total order protocols (*Validity*, *Uniform Agreement*, *Uniform Integrity* and *Total Order*) and does not specifically rely on any other properties like *Prioritization*, it can be isolated from specific total order broadcast implementations and additional semantics offered by them.

References

[1] 2012. JGroups website: http://www.jgroups.org.

[2] 2012. JSR 51: New I/O APIs for the Java Platform, http://www.jcp.org/en/jsr/detail?id=51.

[3] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Intnl. Conf. on Depend. Syst. and Netw. (DSN)*, pages 327–336, Washington, DC, USA, 2000. IEEE-CS.

[4] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: specification and algorithms. *IEEE T. Software Eng.*, 27(4):308–336, April 2001.

[5] Nina T. Bhatti. *A system for constructing configurable high-level protocols*. PhD thesis, Dept. of Comput. Sc., The University of Arizona, December 1996.

[6] Ken Birman and Robert van Renesse. *Reliable distributed computing with the Isis toolkit*. IEEE-CS Press, Los Alamitos, CA, USA, 1993.

[7] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM T. Comput. Syst.*, 5(1):47–76, 1987.

[8] Patrick G. Bridges, Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Supporting coordinated adaptation in networked systems. In *Wshop. on Hot Topics in Operat. Syst. (HotOS)*, page 162, Elmau, Germany, May 2001. IEEE-CS Press.

[9] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *Intnl. Conf. on Distrib. Comput. Syst. (ICDCS)*, pages 635–643, Phoenix, AZ, USA, April 2001. IEEE-CS Press.

[10] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[11] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[12] Danny Dolev and Dalia Malki. The design of the Transis system. In *Intnl. Wshop. on Theory and Pract. in Distrib. Syst.*, volume 938 of *Lect. Notes Comput. Sc.*, pages 83–98. Springer, Dagstul, Germany, September 1994.

[13] Jim Gray. Notes on database operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[14] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[15] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[16] JBoss. JBoss Netty, 2012. http://www.jboss.org/netty.

[17] Sushanta Karmakar and Arobinda Gupta. Adaptive broadcast by distributed protocol switching. In *Intnl. Symp. on Appl. Comput. (SAC)*, pages 588–589, New York, NY, USA, 2007. ACM Press.

[18] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, June 1979.

[19] Xiaoming Liu and Robbert van Renesse. Fast protocol transition in a distributed environment (brief announcement). In *ACM Symp. on Princ. of Distrib. Comput. (PODC)*, page 341, New York, NY, USA, 2000. ACM.

[20] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In *Intnl. Wshop. on Applied Reliab. Group Comm. (WARGC-ICDCS)*, pages 37–42, Phoenix, AZ, USA, 2001. IEEE-CS Press.

[21] Emili Miedes and Francesc D. Muñoz-Escoí. On the cost of prioritized atomic multicast protocols. In *Intnl. Symp. on Distrib. Obj., Middleware and Appl. (DOA)*, volume 5870 of *Lect. Notes Comput. Sc.*, pages 585–599. Springer, Vilamoura, Portugal, November 2009.

[22] Emili Miedes, Francesc D. Muñoz-Escoí, and Hendrik Decker. Reducing Transaction Abort Rates with Prioritized Atomic Multicast Protocols. In *Intnl. Euro. Conf. on Paral. and Distrib. Comput. (Euro-Par)*, volume 5168 of *Lect. Notes Comput. Sc.*, pages 394–403. Springer, Las Palmas de Gran Canaria, Spain, August 2008.

[23] Emili Miedes and Francesc D. Muñoz-Escoí. Dynamic switching of total-order broadcast protocols. In *Intnl. Conf. on Paral. and Distrib. Proces. Tech. and Appl. (PDPTA)*, pages 457–463, Las Vegas, Nevada, USA, July 2010. CSREA Press.

[24] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Intnl. Conf. on Distrib. Comput. Syst. (ICDCS)*, pages 707–710, Phoenix, Arizona, USA, April 2001. IEEE-CS.

[25] José Mocito and Luís Rodrigues. Run-time switching between total order algorithms. In *12th Intnl. Euro-Par Conf. (EuroPar)*, volume 4128 of *Lect. Notes Comput. Sc.*, pages 582–591. Springer, Dresden, Germany, 2006.

[26] Louise E. Moser, P. Michael Melliar-Smith, Deborah A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, April 1996.

[27] Olivier Rütti, Pawel Wojciechowski, and André Schiper. Structural and algorithmic issues of dynamic protocol update. In *Intnl. Paral. and Distrib. Proces. Symp. (IPDPS)*, page 9 pgs, Rhodes Island, Greece, April 2006. IEEE-CS Press.

[28] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software Pract. Exper.*, 28(9):963–979, 1998.

[29] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4):76–83, 1996.

[30] Pawel T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: a framework for a synchronisation-augmented microprotocol approach. In *Paral. and Distrib. Proces. Symp. (IPDPS)*, page 64b, Santa Fe, New Mexico, USA, April 2004. IEEE-CS Press.