Dynamic Total-Order Broadcast Protocol Replacement

Emili Miedes De Elías, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia (SPAIN)

{emiedes, fmunyoz}@iti.upv.es

Technical Report TR-ITI-SIDI-2010/001

Dynamic Total-Order Broadcast Protocol Replacement

Emili Miedes De Elías, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia (SPAIN)

Technical Report TR-ITI-SIDI-2010/001

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

March 31, 2010

1 Introduction

Total-order broadcast protocols are a basic *building block* in order to develop highly available distributed applications, since it is needed for ensuring that the updates to be applied by any given replica are adequately propagated and applied in all other ones. This provides the needed basis for ensuring a *sequential* [18] consistency model, for both the active [32] and passive [6] replication models.

It is accepted that there is no single total order protocol that provides the best performance under any working conditions [12, 11]. 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.

Nevertheless, 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 application designers 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 might become *unsuitable* due to these changes.

Due to this, there should be some mechanism that allows the applications to use, in every moment, the most suitable total order protocol; i.e., a *protocol switching* tool. Suitability may be decided 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 would allow an application to *adapt to* changing working conditions and, in general, to get a better performance.

There already are multiple switching protocols of this kind [21, 8, 29, 28]. Most of them assume that such switches will seldom arise and that consensus is needed in order to accept such expensive switch operation. So, they are structured in two phases, leading to a blocking behavior [33]; i.e., no message can be broadcast from the point when a process has requested the switch until the moment all of them have accepted such switch and have delivered all previously broadcast messages.

We propose a different alternative. In it, all broadcast protocols might be known in advance and they can be plugged into the switching protocol. Switches can be concurrently requested and all of them are accepted, although such requesting order is recorded. To this end, each broadcast message should contain

in its headers which protocol was used to propagate it, and each process should know how many messages should be managed by each running protocols. So, in the switching interval both protocols (old and new) are able to work simultaneously: new messages can be broadcast with the new protocol and the old protocol is still able to deliver its messages. As a result, no blocking interval arise and the switch action is usually faster than in previous protocols. Additionally, this switching mechanism is also able to manage prioritized total-order protocols, whose usefulness has been proven in some of our previous works [26, 27].

The rest of this paper is structured as follows. Section 2 presents our dynamic switching architecture. Section 3 describes the switching protocol and Section 4 analyzes some of the issues that should be faced by that protocol. Later, Section 5 states and proves the properties to be satisfied by the switching mechanism and Section 6 discusses related work. Finally, Section 7 gives the conclusions.

2 A Dynamic Protocol Replacement Architecture

In [25], we presented an architecture for dynamically replacing Group Communication Protocols (GCPs). Such an architecture was designed to allow the dynamic replacement of FIFO or total order broadcast protocols.

In the following sections we review the work presented in [25] and discuss how such an architecture could be adapted to fit the needs presented in Section 1.

In Figure 1 we show a reviewed version of the high level graphical description of the architecture proposed in [25].

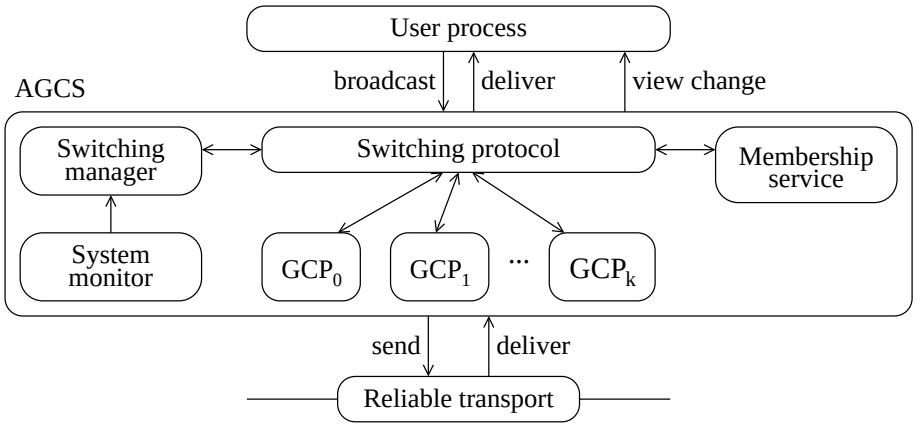

Figure 1: Architecture of a node

This architecture is composed of a main component, called *Adaptive Group Communication System* (*AGCS*). As shown in the figure, the user process sits on top of this architecture 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 also specific components.

The *Switching protocol* implements the mechanism of replacing the GCPs in runtime. 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 measures provided by the *System monitor* and uses them to decide about GCP changes.

The original architecture presented in [25] and the one we present in Figure 1 include 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

In this Section we present the *Switching protocol*. We first provide an overview of the protocol and then we present some notation details and a pseudocode algorithm of the protocol. We finally discuss some details not covered in the first overview.

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 redirected to the current GCP. Incoming messages are received by the current GCP and directly handled to the protocol, which in turns 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 delivered by the new protocol, as it has already been started in all nodes. These messages are not delivered to the user process yet, but queued 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 all 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 algorithm of the protocol is shown in Algorithms 1 and 2.

The protocol uses several *global variables*. k is a counter of the GCP changes. 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 $P_k.k$ 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.

We also assume that the managed GCPs provide a *to–bcasts* primitive to broadcast 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. we assume that 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, respectively, 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 some external 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

In this Section we discuss 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 protocol we are presenting offers a number of advantages over the protocols reviewed in Section 6 and the protocol we proposed in [25].

First of all, our solution 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* immediatelly delivers to the user process the queued messages delivered 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. These have not been included in the protocol algorithm to simplify its presentation.

First of all, 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 can be used to save

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

```
1:  CREATE_P(p,g):
2:      p.GCP ← g
3:      p.k ← next_k
4:      p.sent ← 0
5:      p.other_sent[q] ← 0, for each process q in live_nodes
6:      p.ack_received[q] ← false, for each process q in live_nodes
7:      p.delivered[q] ← 0, for each process q in live_nodes
8:      p.deliverable ← {}
9:
10: INIT(G):
11:     current_k ← 0
12:     next_k ← 0
13:     changing_gcp ← false
14:     instance, prepare and initialize G
15:     call CREATE_P(P_{next_k}, G)
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:
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 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:
```

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

```
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:
```

an 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

Another issue that can be discussed is the ability of the *Switching protocol* to face concurrent starts of the switching procedure. Indeed, in case several protocol switchs are started concurrently by different nodes or even the same node, the use of a total order broadcast protocol to broadcast the PREPARE messages forces that all the nodes receive the same PREPARE messages in the same order.

First of all, 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. In this handler, we just update the local copy of the set of nodes considered alive and call the FINISH_PENDING function. This call is needed because it may happen that the only messages required to finish one or more iterations were sent by processes declared failed. In this call, all the pending iterations are checked, considering only the alive nodes.

The reaction to view changes we present in these algorithms is actually minimum. In Algorithm 3 we extend 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* [9].

If the *Sending View Delivery* property has to be provided, 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.

For this, we propose the following procedure. 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.

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.

To implement these features, we propose a number of changes, in Algorithm 3. First we add two new global variables. The $provide\_sending\_view$ variable is a flag used to know if the *Sending View Delivery* property has to be ensured. It's value is set to the value of the $sending$ parameter of the `INIT` handler. This way, it can be decided externally. If it is set to $false$, then the *Same View Delivery* property is offered instead. Moreover, we use a $changing\_view$ global flag, used to know if there is a view change 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*

In this section we provide some properties of the *Switching protocol* and some reasoning about their correctness. First, we propose some lemas used to prove the properties.

Lema 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).

If we consider the modifications presented in Algorithm 3, then, 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, we have to show 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).

Lema 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 now that a finite number of message 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 Lema 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.

Lema 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 Lema 2, we know that 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 can not 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).

Lema 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 we know that 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 Lema 3 (*Local Integrity*) stated above, the *Switching protocol* eventually delivers the message to the user process.

If a GCP change happens, Lemas 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 *upgoing* 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 Lema 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 Lema 2 (Upwards Validity), the *Switching protocol* eventually delivers up the message to the user process in all correct nodes.

In case 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 Lema 3, we know that the *Switching protocol* can not deliver twice the same message. It delivers a message twice only if the GCP has delivered twice that message to it.

By the *Uniform Integrity* property of the GCP, this can only happen if the GCP in the sender node has broadcast twice the message. By Lema 1 (*Downwards Validity*), we know that this is only possible if the sender node broadcasts twice the same message through the GCP, and this can only happen if the user process in the sender node broadcasts 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 N and N', whether correct or faulty, both deliver messages m and m', then the Switching protocol in N delivers m to its user process before m' if and only if the Switching protocol in N' delivers m to its user process before m'.*

Proof. Let's suppose that the *Switching protocol* in both nodes N and N' deliver two messages m and m'. If N 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 N delivers m using $P_k.GCP$ and delivers m' using $P_{k'}.GCP$, with $k < k'$. Then, N' also delivers m using $P_k.GCP$ and m' using $P_{k'}.GCP$. Moreover, by Lema 4 (*Change Safety*), as m has been broadcast using $P_k.GCP$, N' delivers m to the user process before delivering any other message broadcast by $P_{k'}.GCP$, which means that N' delivers m prior to m'.

The reasoning is also valid if N or N' fail after delivering m and m', respectively. On one hand, N and N' 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 Lema 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 N and N' deliver m before delivering m'.

6 Related Work

In this Section we briefly review some previous work that is related to our concern. The reviewed papers are divided into two different groups. In a first group, we include those papers that propose some configurable architecture or mechanism that is able to adapt to changing environments or settings, by means of *tuning* its behaviour, but without performing structural changes like the ones carried on by a switching protocol. We also include some other work directly related to adaptable systems. In a second group, we include those papers that use some *dynamic switching* mechanism that is able to replace the current implementation of one or several services. The papers in both groups are presented chronologically. Some of the references cited present a solution based on a switching algorithm while others present work related to adaptive systems from a more general point of view.

6.1 Configurable systems

6.1.1 Composability in x-kernel (Hutchinson et al., 1991) and Coyote (Bhatti et al., 1995)

The x-kernel operating system kernel [15] was designed to ease the design and development of network protocols. It is considered one of the first systems based on *composable* stacks of protocols. For instance, it includes protocols that implement different communication standards like IP, UDP, TCP and even low level protocols like ARP.

In x-kernel, the composition of the protocol stacks to use is defined statically, in configuration time. In boot time, each protocol communicates with its underlying protocol in order to agree on the relationship. Once the kernel is booted, there is no mechanism to dynamically change the composition of the protocol stacks used by x-kernel.

The Coyote system [3] is based on x-kernel and proposed the decomposition of regular x-kernel protocols into a set of *micro-protocols*. In configuration time, a first construction step if performed, by combining the micro-protocols of a protocol. Then, a regular x-kernel configuration step is carried, to build regular x-kernel protocol stacks.

As in x-kernel, the configuration of Coyote is static and no dynamic reconfiguration mechanism is available.

6.1.2 ADAPTIVE (Schmidt, 1993)

The ADAPTIVE system [31] is an environment to develop network protocols, designed to adapt to heterogeneous and changing environments. First of all, it offers a number of high-level abstractions to specify the behavior of the network services that will be finally offered to the user application, according to the current setting (e.g. the topology and type of the network) and the application quality-of-service needs. The specifications of the services are used to instantiate *protocol machines*, which are protocol implementations available in a repository and tuned to fit the such requirements and needs. Moreover, in configuration time, user applications can refine the specification of the network protocols to use. The reconfiguration mechanism is then able to tune the current protocol machines or create new ones, in order to adapt to the application needs.

6.1.3 About the use of standard interfaces (Wiesmann et al., 2003)

Several efforts have been made to propose a set of standard interfaces that express a wide range of group communication related services like group membership, or communication primitives.

In [36], the authors propose the use of middleware architectures built up from components that follow standard and well-known interfaces. The architecture they propose can be used to build distributed systems and it includes a membership service, a fault detector service and some messaging services. These services are implemented by components that must offer well-defined and standard interfaces to the components *above them* (i.e., components that *use* the services they offer). These components, in turn, use the services offered by the components *below* them.

The benefit of using standard interfaces is twofold. First, as the knowledge of the standards to use can be reused, the design and implementation of new components is easier and simpler. Moreover, the use of standard interfaces allows the replacement of the implementation of a given component by a new one, as justified in previous sections.

In [36], for each service, several standard alternatives are considered. For instance, TCP/IP UDP/IP, IP-multicast, BEEP, APEX and JMS are considered as standards to define the behavior of the messaging components while LDAP and SNMP standards are considered for the membership service and SNMP and CMIP, for the fault detection service.

6.1.4 A survey of middleware software (Sadjadi, 2003)

In [30], a survey of configurable and adaptive middleware is presented. This work is actually a first version of [24], which is reviewed in a later section. In the survey, a number of solutions are classified into different classes.

The survey first identifies four key technologies that offer *composability* and *adaptability*: a) computational reflection [22], based on the use of introspection, b) component-based design, c) aspect-oriented programming [17] and d) software design patterns.

In a first taxonomy, it classifies middleware software depending on the abstraction layer in which they may be placed: a) *Host-infrastructure*, b) *distribution*, c) *common-services* and d) *domain-services*.

A second classification proposed in [30] classified middleware software according to its adaptation level: a) *configurable*, b) *customizable*, c) *tunable*, and d) *mutable*. The *mutable* class is the only one that can be considered completely dynamic and may include some techniques like introspection, aspect-based programming and dynamic code loading.

Finally, in [30] a third classification is proposed that classifies middleware according to its application domain: a) *QoS-oriented systems*, b) *dependable systems*, and c) *embedded systems*.

In the survey a big number of solutions are reviewed, many of them related to CORBA. Nevertheless, none of them can be compared to the switching mechanism proposed in Section 3 or the other solutions reviewed in this Section.

6.1.5 A taxonomy of compositional adaptation (McKinley et al., 2004)

In [24] (an extended version of [23]), a survey of adaptive systems is presented.

In this survey, two main types of adaptation are identified. *Parameter adaptation* is present in systems that are able to modify the values of their parameters and variables in order to adapt to changes in their settings, environments, working conditions, etc. On the other hand, *compositional adaptation* involves the ability to algorithmically or structurally change a system in order to perform such adaptation. In the survey, a *taxonomy of compositional adaptation* is presented.

The taxonomy is multidimensional. The solutions surveyed are classified according three different criteria: a) how, b) when and c) where to *compose* (i.e. perform a system's adaptation).

Regarding to *how to compose*, several mechanisms can be used.

- Redirection of function pointers. The pointers that point to the functions that contain the code to change or *adapt* can be *redirected* to point to different functions (for instance, *proxy* functions).
- *Wrappers*. The use of the *wrapper pattern* allows business objects to be encapsulated by *wrapper objects* that can control the original objects.

- *Proxies*. According to the *proxy pattern*, some *proxy* code can be *inserted* in the original code, to intercept and manage regular invocations to business logic.
- The *strategy pattern*. Each service implementation is encapsulated under an interface. This allows the replacement of a given implementation by another one, as long as both share the same interface.
- *Virtual components*. A *virtual component* is a placeholder that allows the loading and unloading of service code in an application-transparent manner.
- *Meta-Object Protocols*. A specific protocol can be used to dynamically replace the implementation of a service.
- *Aspect weaving*. Aspect Oriented Programming can be used to *inject* orthogonal functionality to existing service implementations.
- Middleware interception. Regular service requests and the corresponding responses are intercepted at a middleware-level layer. Adaptation can be performed in such layer.
- Integrated middleware. Besides indirectly using a middleware layer, the user applications can also explicitly make use of their services.

Additional criteria are considered regarding this criterion: transparency of the solution, granularity, coverage and support of standards.

The *transparency* criterion expresses the transparency level of the adaptation mechanism respect to the functional code of the application, the adaptive code, the distribution middleware services (if any) and the virtual machine (if any). The *granularity* criterion is useful to know the granularity of the adaptation mechanism (per system, per class, per object, per method or per invocation). The *coverage* criterion distinguishes systems that only are appliable to local invocations from those that also consider remote invocations. Moreover, the ability to apply the adaptive mechanism to just a subset of the invocations is also checked. Finally, the *standards support* criterion allows to know which standard like CORBA/CCM, Java RMI/J2EE and DCOM/.NET are supported by the systems.

Regarding to *when to compose*, two first categories can be distinguished: *static composition* and *dynamic composition*. Static composition is performed in configuration, compilation, deployment, linking or even loading time while dynamic composition, is performed in run-time. Static composition is usually easier to perform but it is usually less flexible and powerful than dynamic composition, which is, on the other hand usually more complex to perform.

As there are different levels of static composition, it can be achieved in different manners. Simpler static composition can be performed by tuning hardwired parameters and code and recompiling the system. More flexible mechanisms perform the adaptation in deploy time, by choosing the proper components and modules to use. The most powerful alternatives include late binding and dynamic class loading.

On the other hand, systems that use dynamic composition can be *tunable* or *mutable*. *Tunable software* can be dynamically configured and *adapted* by run-time tuning some of their parameters and variables. *Mutable software* offers the possibility of altering the functionality of the system, for instance, by dynamicaly replacing the code of some of their components.

Regarding to *where to compose*, middleware-level and application-level alternatives can be considered. Middleware-level alternatives include constructing a layer of adaptable software and attach it to the user application and modifying a virtual machine in order to add some adaptive support. Application-level alternatives imply adapting part of the application itself. Different alternatives exist like the use of programming languages that natively offer some *adaptation support* (like CLOS or Python), the extension of the programming language run-time mechanism used by the application or the use of Aspect Oriented Programming libraries.

In the survey, more than forty solutions are classified according to this taxonomy, including classic Group Communication Systems (Ensemble, Totem and others) and CORBA middlewares (ACE, TAO, CIAO and others).

6.1.6 A standard GCS interface (GORDA project, 2007)

In [7], another proposal is presented. The idea is to have a middleware layer that provides an abstract Application Program Interface to be used by conventional distributed systems. This layer is placed between an application and a Group Communication System, thus acting as an adapter of the latter.

This strategy yields two major benefits. First, it avoids the use of implementation-specific semantics and interfaces. Moreover, it isolates applications from a specific GCS implementation and thus allowing a future replacement of the current implementation. As a side effect, this independence also eases the evaluation of the behavior and performance of an application using different GCS implementations, for instance, in order to choose one of them.

This strategy is implemented in the *Group Communication Service* project [10]. Nowadays, this project includes bindings to existing GCS implementations like Appia, JGroups and Spread and other communication services like an IP-based multicast service and NeEM.

Unfortunately, this middleware architecture can only be statically configured and no dynamic reconfiguration or switching is possible for the moment.

6.2 Dynamic switching systems

6.2.1 Ensemble's *Protocol Switch Protocol* (van Renesse et al., 1998)

The Ensemble system [14] is a group communication system based on the configuration and use of a *stack of protocols*, as in its predecessor Horus [35]. 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 [34], the 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, 19]. In the first phase, one of the participant nodes takes a coordinator role and broadcasts a `FINALIZE` message, to ask to all nodes to start a protocol stack replacement. This message includes the composition of the new protocol stack. Upon reception of the `FINALIZE` message, each node stops the protocols in its current protocol stack, builds up the new protocol stack and then sends back a `FINALIZE-ACK` message to the coordinator. When the coordinator has received all the acknowledgement messages, it starts the second phase.

In the second phase, the coordinator broadcasts a `START` messages. When a node recives the `START` message, it discards its current protocol stack and starts the regular operation with the new protocol stack.

The protocol includes some fault-tolerance support that tolerates the loss of messages (by means of retransmissions) and the node failures or disconnections.

On the other hand, in the coarse description of the protocol in [34] no details are given about the guarantees needed to multicast the `FINALIZE` and `START` control messages. Moreover, nothing is said about the need to block incoming or outgoing messages.

The PSP present 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 protocols of the stack.

6.2.2 Protocol switching based on state transformation (Liu et al., 2000)

In [20], the authors present a mechanism alternative 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 runtime, 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.

6.2.3 Ensemble's second switching protocol (Liu et al., 2001)

In [21] a second *Switching Protocol* (SP) is presented. Unlike the protocol presented in [34], the SP allows the replacement of a single protocol of the Ensemble's protocol stack.

The protocol is presented as a *wrapping* protocol 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 operates in *normal mode*, it just forwards up and down the messages sent by and delivered to its neighbor layers. When operates in *switching mode*, it performs a protocol replacement. As in [34], 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 [34] 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.

When no protocol change is being performed, the nodes forward a token whose mode is `NORMAL`. To start a protocol change, a manager node waits until it receives a `NORMAL` token. Then it changes the mode to `PREPARE` and forwards the token to the next node in the ring. When a node receives a `PREPARE` token, it saves in some field of the token the number of messages it has sent with the current protocol and then forwards the token. When the manager node receives back the `PREPARE` token, it contains the numbers from all nodes. Then, it changes the token to `SWITCH` and forwards it again.

When a node receives a `SWITCH` token, it gets the number of messages sent by each node. When the manager node receives the `SWITCH` token, it changes the token mode to `FLUSH` and then forwards it once more.

When a node receives the `FLUSH` token it waits until it has received all the messages sent by all nodes with the current protocol. Then, it changes the current protocol to the new one and forwards the token. When the `FLUSH` token is finally received by the manager node, the protocol replacement is finished.

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. Indeed, as the new protocol is not started until the third round of the switching protocol, no messages can be sent using the new protocol until then[1]. 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.

To argue about the correctness of the protocol, in [21] the authors formulate six *metaproperties* (*safety*, *asynchrony*, *delayable*, *send-enabled*, *memoryless* and *composable*) which are properties that describe other properties. Then, they argue that the switching protocol *preserves* these *metaproperties*. In short, they argue that if two protocols (for instance, two total order protocols) offer some property *P* (for instance, a *Total Order* property), and P satisfies those six *metaproperties*, then the switching protocol *is* a protocol that in turn offers *P*. This is formally proved in [4], by means of the NuPRL theorem prover [1].

6.2.4 Adaptive architecture in Cactus (Chen et al., 2001)

In [8, 5], another adaptive architecture for run-time protocol switching is proposed. This architecture is designed for Cactus [2], a framework for building distributed protocols and applications.

As in other distributed middlewares and frameworks, a Cactus application is based on a stack of layered components and each one of these offers a service. Some of these components may be *adaptive*, which means that they include 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 adaptive component also 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,

[1] Although it is not explicitly said in [34], it is assumed that once a protocol change is started, i. e. once a node receives a `PREPARE` token, the message sending with the current protocol is stopped.

closely related to the first one, includes the election of the new implementation of the service. As in the solutions proposed by other authors, very little detail about these phases is given.

The third phase is the *adaptation* phase, which in turns consists of three steps: a) preparation, b) *outgoing switchover* and c) *incoming switchover*. This is a general scheme and the basic idea is that any protocol change can be decomposed in these steps, regardless the kind and nature of the service implementations that are to be replaced.

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.

The generic protocol change scheme is implemented in the adaptor module of the adaptive component. This module depends on the semantics and nature of the service implementations to replace. In [8], the replacement of the total order broadcast service is given as an example of a implementation of the general scheme. Basically, the replacemente procedure is the one specified by the general scheme. A significant detail is that once reached the synchronization barrier, at the end of the first step and before performing the *outgoing switchover*, the outgoing messages that could not be sent with the previous total order protocol are broadcast, by means of the new total order protocol.

One of the main drawbacks of the solution presented in [8] is that it forces the service implementations to fulfill a given interface. Actually, this requirement is not too strong, since this adaptive architecture was designed for Cactus systems, that are already forced to follow this requirement. Anyway, such a drawback may be solved by means of additional indirection layers that could be placed on top of each particular service implementation, thus acting as *adaptors*.

In essence, our proposal shares the general idea behind this proposal and its *structure* organized in three different parts. The broadcast of the PREPARE message in the algorithm proposed in Section 3 can be compared to the *synchronization barrier* used in [8]. Moreover, both solutions need to queue the outgoing messages that could not be sent with the previous protocol and send them later, with the new protocol, once it has been activated.

6.2.5 Dynamic Protocol Update (Rütti et al., 2006)

In [29] the problem of *Dynamic Protocol Update* is considered, 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 of the protocol stack used by a user 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 [37] but the basic idea may be applied to other protocol stack-oriented frameworks. The goal of the architecture presented is to allow the dynamic replacement of software components, thus easing the software maintenance and upgrade tasks. Nevertheless, this architecture can also help to improve the performance of the applications, as proposed in Section 1.

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. Although some minor differences exist, the operation of the protocols is basically very similar. For instance, 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. A *service request* is either a consensus instance or the broadcast of a message, depending on the protocol to replace.

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 recives 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. This way, the switching protocol avoids the need of an additional acknowledgment message round (as in other proposals like the presented in this Section or the one presented in Section 3). As in other proposal, nothing is said about how is decided to start a protocol change or which criteria is considered.

In [29], a discussion of the properties guaranteed by the switching protocols is also provided. These properties are expressed in terms of *modules*, *services* and *module bindings*. A protocol stack is modelled as a stack of *modules*. Each module is configured as a provider of a *service* by means of a *module binding*. A binding can be done statically, in configuration time, or dynamically, during a protocol change.

First, two properties of the switching protocols are proposed. Both properties have a *strong* and a *weak* variant. The *stack well-formedness* property expresses the need to have all the services bound to any module. The strong variant of this property requires that if a service is invoked it must have been bound to any module. The weak variant of the property requires that if a service is invoked, it is *eventually* bound to any module.

The *protocol operationability* property requires the need to have the required module installed in a stack when a protocol change is issued. Informally, the strong variant of this property requires that if a module (protocol) is bound in the stack of a node, then the stacks of all nodes contain that module. The weak variant of the property requires that this binding is just eventually done.

In [29] (and also [38]), it is shown that the atomic broadcast switching protocol ensures the *strong stack well-formedness* and the *weak protocol operationability* properties. Moreover, it is also shown that the regular properties of the atomic broadcast protocols (*Validity*, *Uniform Agreement*, *Uniform Integrity* and *Uniform Total Order*) are preserved by the atomic broadcast switching protocol.

Finally, some performance evaluation of both protocols is also presented. This evaluation includes the analysis of the latency of a series of messages broadcast by a set of nodes, during which an atomic broadcast protocol replacement is requested. 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.

6.2.6 Mocito's run-time switching (Mocito et al., 2006)

In [28] another switching protocol for total order protocols is proposed. This is, in essence it is very similar to the one presented in [25].

In particular, they share some relevant features. First, it avoids blocking message sending with the new protocol so the flow of application messages is never blocked. It also sets a point in time from which no more messages are sent with the current total order protocol. Moreover, 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.

They differ in the way the participant nodes *learn* about when they must *deactivate* the current total order protocol. In [25], the nodes count the number of messages broadcast with the current protocol and when a protocol change is started, this information is spread so all nodes know how many messages have to be delivered with the current protocol before deactivating it. In [28], each node broadcast 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.

6.2.7 Broadcast protocol switching (Karmakar et al., 2007)

In [16], the authors deal with 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 Deep-First Search reduces the load on individual nodes when the global network load is higher.

The mechanism discussed in [16] 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. In the paper, a protocol is shown to build a new DFS spanning tree. Nevertheless, no protocol is shown to build a BFS spanning tree.

7 Conclusion

In this paper we review the problem of dynamically replacing the total order broadcast protocol used by a distributed application. As a result, we provide 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.

In this paper we provide 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.

Although this switching protocol was designed to allow the dynamic replacement of regular total order broadcast protocols and the use of prioritized total order protocols is not mentioned, the switching protocol can also be used to replace *prioritized* total order broadcast protocols, without any further modifications.

To argue about this, we must consider that prioritized protocols 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 in any other properties like *Prioritization*, it can be isolated from specific total order broadcast implementations and additional semantics offered by them.

Acknowledgements

This work has been partially supported by EU FEDER and Spanish MICINN under grant TIN2009-14460-C03.

References

[1] Stuart F. Allen, Rich Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In *17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes of Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[2] Nina T. Bhatti. *A system for constructing configurable high-level protocols*. PhD thesis, Department of Computer Science, The University of Arizona, Dec. 1996.

[3] Nina T. Bhatti and Richard D. Schlichting. A system for constructing configurable high-level protocols. In *SIGCOMM*, pages 138–150, 1995.

[4] Marck Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. *Lecture Notes in Computer Science*, 2152/2001:105–120, 2001.

[5] Patrick G. Bridges, Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Supporting coordinated adaptation in networked systems. In *Eigth Workshop on Hot Topics in Operating Systems*, 2001.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. J. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, Wokingham, UK, 2nd edition, 1993.

[7] Nuno Carvalho, José Pereira, and Luís Rodrigues. Towards a generic group communication service. In *Distributed Objects and Applications International Conference (DOA'06)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1485–1502, 2006.

[8] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 635–643, Mesa, Arizona, USA, 2001.

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

[10] The GORDA Consortium. Group communication service in sourceforge.net. http://jgcs.sourceforge.net.

[11] Xavier Défago, André Schiper, and Péter Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

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

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

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

[15] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[16] Sushanta Karmakar and Arobinda Gupta. Adaptive broadcast by distributed protocol switching. In *ACM symposium on Applied computing (SAC'07)*, pages 588–589, New York, NY, USA, 2007. ACM.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, , and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, 1997.

[18] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

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

[20] Xiaoming Liu and Robbert van Renesse. Fast protocol transition in a distributed environment (brief announcement). In *19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, page 341, New York, NY, USA, 2000. ACM.

[21] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luís Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*. IEEE CS Press, 2001.

[22] Pattie Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155, 1987.

[23] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[24] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan 48824, July 2004.

[25] Emili Miedes, Mari-Carmen Bañuls, and Pablo Galdámez. Group communication protocol replacement for high availability and adaptiveness. In *Advanced Distributed Systems: 6th International School and Symposium, ISSADS 2006*, 2006.

[26] Emili Miedes and Francesc D. Muñoz-Escoí. Managing priorities in atomic multicast protocols. In *International Conference on Availability, Reliability and Security (ARES 2008)*, pages 514–519, Barcelona, Spain, 2008.

[27] Emili Miedes, Francesc D. Muñoz-Escoí, and Hendrik Decker. Reducing transaction abort rates with prioritized atomic multicast protocols. In *14th International European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, pages 394–403, Las Palmas de Gran Canaria, Spain, 2008. Also available as a Technical Report ITI-ITE-07-22.

[28] José Mocito and Luís Rodrigues. Run-time switching between total order algorithms. In *EuroPar 2006*, 2006.

[29] Olivier Rütti, Pawel Wojciechowski, and André Schiper. Structural and algorithmic issues of dynamic protocol update. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[30] Seyed Masoud Sadjadi. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan 48824, 2003.

[31] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, 1993.

[32] F. B. Schneider. Replication management using the state-machine approach. In S. J. Mullender, editor, *Distributed Systems*, chapter 7, pages 166–197. Addison-Wesley, Wokingham, UK, 2nd edition, 1993.

[33] Dale Skeen. Nonblocking commit protocols. In *SIGMOD Intnl. Conf. on Management of Data*, pages 133–142, Ann Arbor, Michigan, April 1981. ACM Press.

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

[35] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[36] Matthias Wiesmann, Xavier Défago, and André Schiper. Group communication based on standard interfaces. In IEEE, editor, *2nd IEEE International Symposium on Network Computing and Applications (NCA-03)*, 2003.

[37] Pawel T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: a framework for a synchronisation-augmented microprotocol approach. *18th IEEE Parallel and Distributed Processing Symposium (IPDPS2004)*, April 2004.

[38] Pawel T. Wojciechowski and Olivier Rütti. On correctness of dynamic protocol update. In Springer LNCS 3535, editor, *7th IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS05)*, June 2005.