Dynamic Switching of Total-Order Broadcast Protocols∗

Emili Miedes Francesc D. Muñoz-Escoto†
Instituto Universitario Mixto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia, Spain
{emiedes, fmunyoz}@iti.upv.es

Abstract

The exchange of total order broadcast protocols for dependable applications has been already studied and solved in several previous papers. To this end, some switching mechanism is needed. All these solutions have been inspired in a two-phase algorithm. The first phase is needed to agree on the need of change, stopping then the old protocol. Whilst the second one is needed to wait for the delivery of all previously broadcast messages, allowing to start later with the new protocol. This introduces a blocking interval that could be quite long in networks with high latency (e.g., WANs) or in systems with slow (i.e., heavily loaded) processes. We propose a new switching mechanism that avoids all such inconveniences, ensuring a fast migration between different broadcast protocols.

KEYWORDS: Broadcast protocol, reliable broadcast, atomic broadcast, switching protocol, group communication system († Contact author)

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 [6] consistency model, for both the active [15] and passive [1] replication models.

According to [5], several different families of total-order broadcast protocols exist: fixed sequencer, moving sequencer, privilege-based, communication history, and destinations agreement. It is accepted that there is no single total order protocol family that provides the best performance under any working conditions [4]. 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 chosen protocol might become unsuitable due to these changes.

As a result, 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 available protocols 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 [7, 2, 14, 13]. 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 [16]; 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 protocol. 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 arises 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 [10, 12, 11].

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. Finally, Section 5 gives the conclusions.

2 A Dynamic Protocol Replacement Architecture

In [8], 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 sequel, we discuss how such an architecture could be adapted to fit the needs presented above; i.e., to implement a non-blocking switching mechanism. Figure 1 shows our system architecture.

It 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.

Our 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 presents 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.

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–beasts 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–beasts 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, P0 would be the struct for the first GCP used, P1 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 Pk, the expression Pk.GCP is used to reference that GCP. The Pk.k field is the number of the replacement by which the Pk.GCP is installed. In general, Pk.k = k. Pk.sent is the number of user messages that have been broadcast by Pk.GCP. Pk.other_sent is an array that stores the number of messages sent by all the processes in iteration Pk.k by means of Pk.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 Pk.other_sent[q] and it is initialized to 0. Pk.delivered is an array that stores the number of messages sent by all the processes delivered by the local process by means of Pk.GCP. Pk.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 Pk.GCP. Pk.deliverable is a list of messages delivered to the protocol by Pk.GCP. If Pk.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 Pk.deliverable, until all the messages sent with all the previous GCPs are delivered.

We also assume that the managed GCPs provide a to–beasts 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
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. Additionally, due to space constraints, a detailed set of correctness arguments and proofs have been omitted in this paper, but they can be found in [9].

4.1 Normal Operation

The protocol we are presenting offers a number of advantages over the protocols cited above. 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. So, 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, nor by networks with high latency (that extend the blocking interval of traditional switching protocols).

Even more, the delivery of messages to the user process is neither blocked. Indeed, when the old protocol is finally disabled, the *Switching protocol* immediately 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 ensuring a correct behavior, 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.

---

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

1. CREATE_P(p, g):
2. \( pGCP \leftarrow g \)
3. \( pk \leftarrow nextk \)
4. \( p.sent \leftarrow 0 \)
5. \( p.ack_sent[q] \leftarrow 0 \)
6. \( p.ack_received[q] \leftarrow false \)
7. \( p.ack_sent[q] \leftarrow 0 \)
8. \( p.ack_received[q] \leftarrow false \)
9. \( p.ack_sent[q] \leftarrow 0 \)
10. \( p.ack_received[q] \leftarrow false \)
11. \( p.ack_sent[q] \leftarrow 0 \)
12. \( p.ack_received[q] \leftarrow false \)
13. \( p.ack_sent[q] \leftarrow 0 \)
14. \( p.ack_received[q] \leftarrow false \)
15. \( p.ack_sent[q] \leftarrow 0 \)
16. \( p.ack_received[q] \leftarrow false \)
17. \( p.ack_sent[q] \leftarrow 0 \)
18. \( p.ack_received[q] \leftarrow false \)
19. \( p.ack_sent[q] \leftarrow 0 \)
20. \( p.ack_received[q] \leftarrow false \)
21. \( p.ack_sent[q] \leftarrow 0 \)
22. \( p.ack_received[q] \leftarrow false \)
23. \( p.ack_sent[q] \leftarrow 0 \)
24. \( p.ack_received[q] \leftarrow false \)
25. \( p.ack_sent[q] \leftarrow 0 \)
26. \( p.ack_received[q] \leftarrow false \)
27. \( p.ack_sent[q] \leftarrow 0 \)
28. \( p.ack_received[q] \leftarrow false \)
29. \( p.ack_sent[q] \leftarrow 0 \)
30. \( p.ack_received[q] \leftarrow false \)
31. \( p.ack_sent[q] \leftarrow 0 \)
32. \( p.ack_received[q] \leftarrow false \)
33. \( p.ack_sent[q] \leftarrow 0 \)
34. \( p.ack_received[q] \leftarrow false \)
35. \( p.ack_sent[q] \leftarrow 0 \)
36. \( p.ack_received[q] \leftarrow false \)
37. \( p.ack_sent[q] \leftarrow 0 \)
38. \( p.ack_received[q] \leftarrow false \)
39. \( p.ack_sent[q] \leftarrow 0 \)
40. \( p.ack_received[q] \leftarrow false \)
41. \( p.ack_sent[q] \leftarrow 0 \)
42. \( p.ack_received[q] \leftarrow false \)
43. \( p.ack_sent[q] \leftarrow 0 \)
44. \( p.ack_received[q] \leftarrow false \)
45. \( p.ack_sent[q] \leftarrow 0 \)
46. \( p.ack_received[q] \leftarrow false \)
47. \( p.ack_sent[q] \leftarrow 0 \)
48. \( p.ack_received[q] \leftarrow false \)
49. \( p.ack_sent[q] \leftarrow 0 \)
50. \( p.ack_received[q] \leftarrow false \)
51. \( p.ack_sent[q] \leftarrow 0 \)
52. \( p.ack_received[q] \leftarrow false \)
53. \( p.ack_sent[q] \leftarrow 0 \)
54. \( p.ack_received[q] \leftarrow false \)
55. \( p.ack_sent[q] \leftarrow 0 \)
56. \( p.ack_received[q] \leftarrow false \)
57. \( p.ack_sent[q] \leftarrow 0 \)
58. \( p.ack_received[q] \leftarrow false \)
59. \( p.ack_sent[q] \leftarrow 0 \)
60. \( p.ack_received[q] \leftarrow false \)
61. \( p.ack_sent[q] \leftarrow 0 \)
62. \( p.ack_received[q] \leftarrow false \)
63. \( p.ack_sent[q] \leftarrow 0 \)
64. \( p.ack_received[q] \leftarrow false \)
65. \( p.ack_sent[q] \leftarrow 0 \)
66. \( p.ack_received[q] \leftarrow false \)
67. \( p.ack_sent[q] \leftarrow 0 \)
68. \( p.ack_received[q] \leftarrow false \)
69. \( p.ack_sent[q] \leftarrow 0 \)
70. \( p.ack_received[q] \leftarrow false \)
71. \( p.ack_sent[q] \leftarrow 0 \)
72. \( p.ack_received[q] \leftarrow false \)
73. \( p.ack_sent[q] \leftarrow 0 \)
74. \( p.ack_received[q] \leftarrow false \)
75. \( p.ack_sent[q] \leftarrow 0 \)
76. \( p.ack_received[q] \leftarrow false \)
77. \( p.ack_sent[q] \leftarrow 0 \)
Algorithm 2 Switching protocol (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+ = \( P_j.\text{other}_{sent}[q] \)
90:    totalDelivered+ = \( P_j.\text{delivered}[q] \)
91: end for
92: if totalOtherSent == totalDelivered then
93:    return true
94: else
95:    return false
96: end if

Algorithm 3 Switching protocol (part III)

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

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 metadata. One of these headers can be used to save 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 is the ability of the Switching protocol to face concurrent starts of the switching procedure. In such case, the use of a total order broadcast protocol to broadcast the PREPARE messages forces that all the nodes receive such 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 \( P_{next,k}.\text{deliverable} \) 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 service. 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 mes-
sages 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 [3].

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. 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. Its 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 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. This protocol admits concurrent starts of the switching procedure.

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 into an architecture like the one presented here. 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 maintained. Moreover, as the switching protocol only relies in the regular properties offered by common total order protocols 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.

References