Adding Priorities to Total Order Broadcast Protocols

Emili Miedes, Francesc D. Muñoz-Escoí

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

{emiedes,fmunyoz}@iti.upv.es

Technical Report TR-ITI-ITE-07/23 (supersedes TR-ITI-ITE-07/07)

Adding Priorities to Total Order Broadcast Protocols

Emili Miedes, Francesc D. Muñoz-Escoí

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

Technical Report TR-ITI-ITE-07/23 (supersedes TR-ITI-ITE-07/07)

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

October, 2007

Abstract

Group communication and total order topics have been studied for more than two decades from both a theoretical and a practical point of view. Most of this work is concerned about the *classical* definition of total order (informally, all the messages are received in the same order). However, in some cases additional guarantees are needed, like optimistic delivery or priority-based delivery, which allows a user application to prioritise the sending or even the delivery of certain messages. In this paper, we present several techniques to modify an existing total order protocol to take into account message priorities and show how existing total order algorithms can be modified according to these techniques.

KEYWORDS: Priority-based broadcast, total order broadcast, group communication protocols.

1 Introduction

Group communication and total order topics have been studied for more than two decades from both a theoretical ([24, 16, 6, 5, 22]) and a practical ([12, 11, 44, 21, 33, 4, 31, 1]) point of view.

Most of this work is concerned about the *classical* definition of total order [24, 16] (informally, all the messages are received in the same order in all their destinations). However, in some cases additional issues have been considered, like optimistic delivery [36, 38, 45].

Another useful characteristic a group communication protocol may offer is priority-based delivery [42, 39, 34], which allows a user application to prioritise the sending or even the delivery of certain messages.

Such a service can be used in a scenario like the following. Consider an application that runs on top of a database replication system and is physically distributed among several sites. Such systems usually follow a *constant interaction* model [48], according to which, updates made by a transaction are broadcast in total order to all the replicas of the database at the end of the transaction, in commit time, using a single message (or, at most, two messages like in the *weak-voting* [49] approach, but only the first one transfers transaction updates). The final order in which a set of messages corresponding to different transactions are delivered by the replicas determines the final order in which a set of updates from the corresponding transactions are applied to the database. This order has a deep impact on the evaluation of the integrity constraints defined in the database. The idea is to alter the order in which transactions are committed for achieving a favorable constraint evaluation, thus reducing the transaction abort rate. Note that other database replication protocols can use a *linear interaction* [48] principle; i.e., multiple update messages per transaction, but their long completion time and their additional recovery problems [19] do not advise their adoption. Note also that the database replication protocol is able to know which database tables and fields have been accessed by a given transaction, and it is able to use such information for assigning priorities. To do so, the replication protocol should be also aware of the semantic integrity constraints defined in the database schema. MADIS [25] is an example of database replication middleware where all these issues can be managed. A transaction implementation based on stored procedures is another alternative for providing all the information needed by the replication protocol in order to assign priorities (accessed tables and fields, values being used in the updates, ...).

Examples of applications that use semantic constraints in their transactions are many, although not all of them have a heavy load of transactions in a short interval of time. Priorization is important when a heavy load is forecast. A possible example could be the application needed for managing the selling of tickets for important sport events. Many people will request such tickets, and they usually are sold in a short interval. Some possible semantic constraints could be to favor requests that only demand a few tickets, against requests demanding many (in order to avoid re-selling). Additionally, a constraint may exist in order to prevent the same individual to send more than two requests for tickets (again, for preventing re-selling). The first constraint needs to assign the highest priority to requests for one single ticket, progressively reducing the priority as the amount increases, and directly rejecting those requests that exceed a certain threshold (e.g., five tickets). This only relies on priorities. The second constraint demands requests authentication, recording their sender, and need to be evaluated once the request has been delivered. Priorities need not be used for this second issue.

This priority-based total order broadcast service can also be used in other scenarios. It can be used to prioritise different types of data traffic, in different scenarios, for different purposes. For instance, consider a distributed application that controls the operation of some facilities that include both critical and non-critical remote systems, by means of control commands issued to them. Messages sent to critical systems can be prioritised over messages sent to non-critical systems, by means of a priority-based total order broadcast protocol.

In this paper, we present a study about the adaptation of *classical* total order protocols to take into account message priorities. The paper is organised as follows. In sections 2, 3 and 4, we show the system model we are assuming, the *ideal* properties of a priority-based total order protocol and some common problems related to priority-based total order protocols, respectively. In section 5, we review the classification of total order broadcast protocols given in [22]. In section 6 we propose several techniques to modify the different classes of total order protocols to take into account the priorities of the messages. We then show, in section 7, the pseudocode of some modified total order protocols. In section 8, we present a few references about priority-based total order, classify them according to the taxonomy of [22] and then compare them against the modifications proposed in section 5. In section 9, we discuss the convenience of applying a middleware-level approach to solve the priority-based total order broadcast problem instead of relying on an application-level solution. We conclude the paper in section 10 with some conclusions and talk about our future plans.

2 System Model

The system we consider is composed of a set of processes $\Pi = \{p_1, p_2, ..., p_n\}$. Processes communicate through message passing by means of a *fair lossy channel*[1].

Each process has a multilayer structure. Figure 1 shows the structure corresponding to the first example of section 1. The user level is represented by a distributed user application that accesses a replicated DBMS which in turn uses the services offered by a group communication system (GCS), that is composed of one or more group communication protocols (GCP). The GCS sends to and receives messages from the network and delivers them to the replication middleware according to some guaranties (for instance, according to some total order). On top of the replication middleware, a user application interacts with the replicated database, for example by issuing transactions composed of read and write operations. As each process may be issuing its own transactions, consistency among all the replicas of the database must be ensured, in spite of the conflicts that may appear among the different transactions that are running in a given moment.

We consider closed groups. A closed group is a group in which every message sender is also a destination, so no external processes are allowed to send messages.

[1]Informally, a fair lossy channel is a channel that is subject to the loss of messages, due to network issues like node disconnections or network partitions, process failures or other reasons. However, a fair lossy channel does not lose all the messages, does not produce new spurious messages, does not duplicate messages and does not change the contents of the messages.

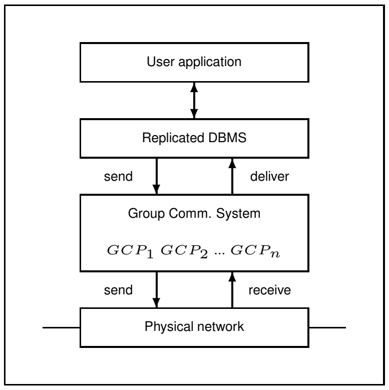

Figure 1: Multilayer architecture.

The system is partially synchronous [20]. Although several definitions exist on partial synchrony, we are considering that on the one hand, processes run on different physical nodes and the drift between two different processors is not known. On the other hand, the time needed to transmit a message from one node to another can be bounded.

Network partitions may also occur. Nevertheless, since we are focusing on the comparison of prioriza-tion techniques we are not addressing these issues here. An implementation of these techniques may rely on some mechanisms like group membership services and fault-tolerance protocols to take care of them.

Messages have an additional property or field that allows a user application to set the *priority level* of the message. The priority $P(m)$ of a message m is a discrete integer number that belongs to a known and bounded interval. Any convention can be used to relate numbers with levels of priority. For instance, we asume that higher values of $P(m)$ correspond to higher priorities and lower values of $P(m)$ correspond to lower priorities. This means that if a message $m1$ has a higher priority than a message $m2$, then $P(m1) > P(m2)$.

3 Properties

Ideally, a priority-based total order broadcast protocol should offer well defined properties regarding the order in which high-priority messages are delivered respect to low-priority messages. Informally, such a property should guarantee that high-priority messages are delivered *first* and low-priority messages can then be delivered.

Different approaches are taken in [39] and [35]. For instance, in [39], a *Priority accounting* property is stated. According to this property, if a message of a given priority has not been delivered at any process, when a message of a higher priority is received, then the latter will be delivered prior to the former. A *priority-based total order multicast protocol* is defined as a broadcast protocol that preserves the *Priority accounting* property in addition to a regular *Total order* property.

In [35], a different approach is taken. Time is divided in *time parts*, and the protocol ensures that all the messages that belong to the same *part* are totally ordered according to their priorities. This time-based solution is also used to avoid starvation of low-priority messages.

4 Problems

Priority-based ordering usually undergoes two common problems: *starvation* ([35]) and *priority inversion* ([41, 7, 14, 47, 46]).

Starvation happens when the delivery (or even the sending) of a low priority message is delayed for a long period of time due to, for instance, a flow of high priority messages. For instance, consider the following scenario. A process sends a constant flow of high priority messages. Another process sends a constant flow of low priority messages. Some priority-based total ordering protocol is ran. The protocol decides that high priority messages must always be delivered prior to any available low priority message. As there is a constant flow of high priority messages, low priority messages never get a chance to be ordered. If no special care is taken in such a situation, low priority messages are never delivered.

However, due to the nature of the application we expect to run over such a priority-based total order broadcast protocol, the starvation problem is not addressed in this work. Several solutions can be applied to at least minimise the impact of message starvation. For instance, in [34] a time division technique is used to avoid message starvation, without introducing too much *priority inversion*, which is the other common issue to care about (see section 8). Moreover, dynamic priority scheduling techniques [37] used by operating system process schedulers may be useful to prioritise too old low priority messages. A common solution consists in dynamically increase the priority of a message if it has been waiting too much time in the sending queue.

Priority inversion happens when high priority messages are forced to wait until some lower priority messages are delivered. This is a typical effect of applying a very strict solution to avoid starvation. In the scenario described above, consider the following solution. The protocol may be modified to force the ordering of a message that is *older enough*. With such a solution, messages that have been waiting *too much* time are ordered as soon as possible. If there are some high priority messages ready to be ordered, they are forced to wait until low priority messages are ordered, (i. e., high priority messages have to wait because *priorities have been inverted*).

In [41, 7, 14], priority inversion is addressed when scheduling the access processes make to certain resources. In [47, 46], the *group priority inversion* is addressed in the context of actively replicated database applications that run in timed asynchronous systems. In that work, priority inversion affects complete user requests, rather than single messages. Priority inversion is also addressed in [2], in the context of real-time database transactions.

5 Reviewing Total Order Protocols

In [22], a survey of total order protocols is given. Such work classifies total order protocols in five different classes: *fixed sequencer*, *moving sequencer*, *privilege-based*, *communication history* and *destinations agreement* protocols.

In the following sections we review that classification and propose how to enable the given classes of protocols to totally order messages in a priority-based manner.

5.1 Fixed Sequencer Protocols

In a *fixed sequencer* protocol, a single process is in charge of ordering the messages. If no processes fail, this special process is fixed. There are three different versions of the *fixed sequencer* basic protocol, as identified in [22], which extends the classification in [26]: the Unicast-Broadcast (UB) protocol, the Broadcast-Broadcast (BB) protocol and the Unicast-Unicast-Broadcast (UUB) protocol.

In the UB version, when a sender wants to broadcast a message to a set of processes, it first sends (unicasts) it to the sequencer. Other processes can concurrently send their own messages to the sequencer and it receives them in some order. When the sequencer receives a message, tags the message with a global sequence number and broadcasts it. All the processes deliver the messages in the order set by the sequence number of each message.

In the BB version, a sender broadcasts the message to all the processes. The sequencer receives different messages broadcast from different senders, in some order. Every message is tagged with the sender identifier and a sequence number local to its sender. When the sequencer receives a new message, it assigns the message a global sequence number and sends a special message containing the original sender identifier and local sequence number, and the global sequence number. All the processes deliver the messages according to the global sequence numbers sent by the sequencer.

In the UUB version, a sender sends a request message to the sequencer, which answers with a sequence number. The sender then tags the message to send with that sequence number and broadcasts it. All the processes deliver the messages in the right order, depending on their sequence number.

In [26], a comparison of the UB and the BB protocols (their original names are PB and BB) can be found.

5.2 Moving Sequencer

In a *moving sequencer* protocol, sequencing is performed by a single process, as in a *fixed sequencer* protocol, but in this case the sequencer is not a fixed process. The sequencer role is transferred from one process to another, among a set of processes that can be the whole set of processes in the system or just a subset.

In several implementations [22], all the processes in the system form a logical ring and the sequencer role is transferred along the ring by means of some kind of token message.

The actual method to order the messages can be any of the methods used by a fixed sequencer (UB, BB or UUB), but some details must be considered when merging the UB, BB or UUB ordering methods with the ring-based method of transferring the sequencer role.

For instance, the combination of the UB method and a logical ring presents an important disadvantage. Whenever the sequencer role is to be transferred to another process, the current sequencer must send to the new sequencer the whole set of incoming messages that have not been sequenced yet. If there are too many pending messages, this transfer may involve a significant bandwidth cost and impose a significant delay in the regular message ordering.

5.3 Privilege Based Protocols

In a *privilege based* protocol, processes can only send messages when they are allowed to do it. If just one process is allowed to send messages at every moment, then the total order can easily be set using just a global sequence number.

Common implementations (e.g. [33, 4]) use a logical ring composed of all the processes in the system. A special message or token is sent along the ring. Only the owner of the token is allowed to send messages. A process that wants to send some messages must wait until it receives the token.

The token contains a global sequence number. Before sending a message, the current token holder tags it with the current sequence number in the ring and then increments it, so the next message, sent by the current sender or by another one, gets the next sequence number. Once the sender has sent its messages, forwards the token to the next process in the ring. Processes deliver the message according to the sequence number set in it.

This protocol is restricted to closed groups (i. e., all the senders know each other), which fits our system model. Some kind of static configuration or membership service is needed.

Moreover, some kind of flow control is needed, to ensure that processes do not send too much messages in its turn and do not keep the token for too much time.

5.4 Communication History Protocols

In a *communication history* protocol, processes use historical information about message sending, reception and delivery to totally order messages.

In [22], two different types of *communication history* protocols are identified: *causal history* protocols and *deterministic merge* protocols.

5.4.1 Causal History Protocols

The class of *causal history* protocols is based on the total order mechanism proposed in [28]. The idea is to causally order messages tagged with Lamport clocks, and extend this causal order into a total order. Although causal order imposes a partial order on the messages that are logically dependent, it is not enough to totally order concurrent messages (informally, messages that are causally independent). These are ordered using the identifier of the message sender.

5.4.2 Deterministic Merge Protocols

In a *deterministic merge* protocol, messages are also broadcast with some kind of timestamp, but unlike *causal history* protocols, these timestamps do not reflect causal relations among messages. In practice, this timestamp can even be a local sequence number. On the other hand, receivers use some local deterministic mechanism to totally order the messages.

In [22], several *deterministic merge* protocols are presented ([15, 27, 3, 17, 18, 10]), although most of them impose significant constraints on the system they may be used in (there are protocols for synchronous systems, protocols that depend on physical clocks or even on redundant reliable channels) so we are not considering them.

A particular deterministic merge protocol In [22], a couple of *deterministic merge* protocols ([9, 8]) based on a round-robin algorithm are cited. As they are good examples of the *deterministic merge* class of protocols, we present here how they can be adapted to consider properties when deciding the total order of the messages.

According to those algorithms, all the processes send a constant flow of messages (that may be *dummy* messages if the user application does not send enough messages), that are tagged with a local sequence number. In each round of the protocol, every process waits for and delivers a message from the first process, then another one from the second process and so on. Once a process has received and delivered a message from all the processes in the system, another round is started. As all the processes are deterministically ordered (by its process identifier) and all the messages sent by a process are also deterministically ordered (by its local sequence number), total order can be easily guaranteed.

This protocol works in a static scenario, in which the set of processes is fixed and their identities are well-known. For this protocol to work in a more dynamic environment, some additional group membership support is needed, to allow a node to know about processes that join the group, leave the group or fail.

On the other hand, this protocol only makes an efficient use of the network when all the nodes have a constant flow of messages to send. If not, dummy messages need to be sent, thus wasting network bandwidth.

5.5 Destinations Agreement Protocols

In a *destinations agreement* protocol, some kind of agreement protocol is run to decide the order of one or more messages.

In [22], three subclasses of *destinations agreement* protocols are identified, according to the type of agreement performed: (1) agreement on the order (sequence number) of a single message, (2) agreement on the order (sequence numbers) of a set of messages and (3) agreement on the acceptance of an order (sequence numbers) of a set of messages, proposed by one of the processes.

5.5.1 Destinations Agreement on the Order of a Single Message (Subclass 1)

An example of such a protocol is an algorithm originally proposed by Skeen and later modified and formalised in [12]. In this protocol, message ordering is performed in three phases (including a broadcast phase). First of all, a process broadcasts a message. Receivers propose a sequence number, and also send their own process identifiers. The sender collects all the proposed sequence numbers and then deterministically decides the final sequence number. Process identifiers are used to break ties, in case that two or more processes propose the same local identifier.

5.5.2 Destinations Agreement on the Order of a Set of Messages (Subclass 2)

An example of this class of protocols is the protocol proposed in [13]. This protocol tries to run a sequence of consensus runs. Each of them is used to agree on the order of a set of messages. All the messages whose order is decided in a consensus run are delivered before any message whose order is decided in the next run. In a particular run, total order is decided in a deterministic manner (for instance, according to the order imposed by the message identifiers, using process identifiers to break ties).

In such a protocol, besides the consensus protocol itself, a reliable message transport is needed. A mechanism to decide the scope of each consensus run (i. e. which messages are ordered by the current consensus run and which ones are ordered by the next one) is also needed.

5.5.3 Destinations Agreement to Accept a Suggested Order (Subclass 3)

An example of this class of protocols is the protocol described in [29]. According to this protocol, when a process broadcasts a message, every process locally saves it in a list of incoming messages. Periodically, some *starter* process decides to start a new consensus run to order a set of messages. To take part in a consensus run, a process sends to the starter process some information about the messages contained in its incoming list. The started process waits for, at least, a majority of responses and then decides a total order for the set of messages. This proposed order is sent to all the processes, which vote to accept or reject it. If the starter process receives a majority of positive votes, the order is accepted and applied by all the processes.

6 Adding Priority Management to Total Order Protocols

We have identified four basic techniques for adding priority management to total order protocols, mostly depending on the point in the life-cycle of the messages in which priorities are considered. These techniques may be called *priority sequencing*, *priority sending*, *priority delivering* and *priority-based consensus*, respectively, and are explained in the following sections.

6.1 Priority Sequencing

Priority sequencing may be applied to sequencer-based total ordering protocols like fixed sequencer (section 5.1) or moving-sequencer protocols (section 5.2). As the name suggests, the priorities of the messages are taken into account when the sequencer is about to sequence each message.

The idea is to keep a list of incoming items. These items may be the messages themselves (as in the fixed or moving UB and BB sequencer protocols) or requests to the sequencer to get a sequencer number (as in the fixed or moving UUB sequencer protocol)[2]. Prior to sending the item, the sender tags it with a priority. According to this priority, the item is inserted in its proper place in the list, so the list is ordered by priority.

To sequence the next item (a message or a request), the sequencer just gets the first available message in the incoming list, this is, the item with highest priority, and sequences them. The meaning of *sequencing an item* depends on the particular version (UB, BB or UUB) of the protocol, as explained in sections 5.1 and 5.2.

This scheme is quite simple but low priority messages may undergo a starvation problem (see section 4) that can be solved using a periodic timeout. When the timeout expires, the sequencer can block the reception of incoming items, assign sequence numbers to all the items currently in the list, broadcast all of them in decreasing order of priority and finally unblock the reception of incoming items. This way, low priority items have a chance of being finally sequenced.

6.1.1 The Cost of Priority Sequencing

The priority of an item (a message or a request) is just an integer so it does not impose a significant overhead in the size of the items and no significant overhead in the use of the network is appreciated.

On the other hand, there is a computational cost related to the reception of the items by the sequencer. Every time a new item is received by the sequencer, it must be inserted in its right place in the list of incoming items, which is ordered in decreasing order of priority. This insertion has a cost in time which is linear to the current size of the list, which in turn depends on several factors, like the sending rate of all the processes and the throughput of the sequencer (number of items sequenced per time unit).

[2]UB, BB and UUB stand for *Unicast-Broadcast*, *Broadcast-Broadcast* and *Unicast-Unicast-Broadcast*, respectively, according to [23].

6.2 Priority Sending

Priority sending differs from priority sequencing in that the priorities of the messages are taken into account in the moment the messages are sent. This kind of modification applies to privilege-based protocols (section 5.2), some protocols of the *deterministic merge* subclass of the *communication history* protocol class (section 5.4.2) like the one presented in section 5.4.2 and the first class of *destinations agreement* protocols (see section 5.5.1), presented in [22].

The idea behind this kind of modification is to send messages in a priority-based order. To this end, each node has a priority-ordered list of outgoing messages quite similar to the one used in section 6.1.

Each outgoing message is placed in its right position in the list, according to its priority. Messages are taken from the head of the list and sent. They then may be treated according to the final protocol used to totally order the messages.

As messages are sent according to the order set by their priorities, if a sender has several outgoing messages pending to be sent, it may send first the most priority one. This also means that low priority messages are retained in the last places of the list, so a starvation problem may occur. A solution similar to the one proposed in 6.1 may be applied here if needed.

6.2.1 The Cost of Priority Sending

The cost of adding priorities by means of this modification is similar to that in section 6.1.

6.3 Priority Delivering

The protocols of the *causal history* subclass of the *communication history* class of [22] can be modified to order messages according to the priorities of the messages.

In the original protocol, causal timestamps are used to causally relate messages. These timestamps are enough to totally order causally dependent messages. Concurrent messages, this is, those that are not causally dependent on each other, are totally ordered by means of a deterministic mechanism, that usually makes use of the identifier of the sender and the sequence number of the message (local to its sender).

The priority delivering modification we propose consists in taking into account the priority of concurrent messages prior to any other criteria. Note that causally dependent messages must still be ordered according to the causal relation imposed by their timestamps, in spite of their priorities, because the modified protocol must still provide the same causal and total order guarantees provided by the original protocol.

According to this, if some node broadcasts a low priority message and then broadcasts a high priority message, as the first one is causally precedent to the second one, the delivery of the latter must be delayed until the first is delivered, regardless of their priorities.

In the original protocol, causally dependent messages are totally ordered by means of their causal timestamps. Concurrent messages are ordered by means of a local deterministic mechanism, applied in delivery time. As the modification we propose is used to modify the way concurrent messages are totally ordered, it is also applied in delivery time.

To solve this issue, priority delivering can be combined with priority sending. This way, messages are sent according to their priorities and as causality is enforced by the delivering mechanism, the local priority-based order is ensured. Moreover, according to the priority delivering modification, concurrent messages are also ordered according to their priorities.

6.3.1 The Cost of Priority Delivering

As in the previous modifications proposed, no significant overhead is imposed on message traffic. The computational cost is similar to that in other classes of protocols and basically depends on the message sending rates.

The proposed modification imposes no significant overhead in the size of the messages. The computational cost mainly depends on the message sending rate. Whenever a message is received by a processor, it is inserted in its right place in the list of incoming messages, depending on its timestamp, the priority and the sender identifier of the incoming message and the existing messages.

	Modification				Who orders			
Protocol	PQ	PS	PD	PC	Q	S	L	O
fixed UB	x				x			
fixed BB	x				x			
fixed UUB	x				x			
mov. UB	x				x			
mov. BB	x				x			
mov. UUB	x				x			
priv-based		x				x		
c.c-causal h.			x				x	
c.c-det m.		x					x	
dest. agr. 1		x				x		
dest. agr. 2				x			x	
dest. agr. 3				x				x

Table 1: A visual classification of total order protocol classes

The cost of this insertion is linear to the number of messages currently in the list, but to be precise, part of this overhead was already present in the original version of the protocol (that corresponding to the comparison of the timestamp of the incoming message against the timestamps of other messages in the list).

Anyway, the size of the incoming list of a process directly depends on the sending rate of the processes in the system and the *productivity* of the local process (measured in messages locally ordered per time unit).

6.4 Priority-based Consensus

To end this classification of modification techniques, we present the *priority consensus*, which is applicable to the second and third classes of *destinations agreement* protocols (see sections 5.5.2 and 5.5.3), presented in [22].

The modification, which is actually quite similar to that of section 6.3, consists in taking into account the priorities of the messages, prior to other criteria, to reach the consensus about the order of a set of messages.

6.4.1 The Cost of Priority Consensus

This modification imposes no significant overhead on message traffic. The computational overhead is also low and, as in other cases, it directly depends on the message sending rates of all the nodes. Moreover, in order to perform the consensus, some additional memory space for message buffering purposes may be needed.

6.5 A Visual Classification

In table 1, we show, in a visual format, which modification type corresponds to each of the total order protocol classes (and subclasses) presented in [22]. We also show, for each protocol class, which is the agent that decides the total order.

The *Modification* keys PQ, PS, PD and PC correspond to *priority sequencing*, *priority sending*, *priority delivering* and *priority-based consensus*, respectively. The *Who orders* keys Q, S, L and O correspond to *sequencer*, *sender*, *local node* and *other*, respectively.

7 Algorithms

In this section we show four algorithms that implement the priority-based total order broadcast service. Each algorithm corresponds to each of the four techniques presented in section 6. These algorithms are just sketches and have not been formally proved. Instead, we just try to illustrate the four techniques of modifying existing total order broadcast protocols.

The algorithms shown are sketches that are not fault-tolerant, since we are not addressing fault-tolerance in this paper. With these algorithms we intend to show the basic protocols and how they can be modified, in the same way of [23], discarding additional issues like fault-tolerance mechanisms. Nevertheless, any solution to address fault-tolerance in similar protocols may be easily applied to the sketches presented here. This approach allows us to compare how the basic protocols operate and perform, without the influence of additional tasks and mechanisms. This way we can focus on comparing the priorization techniques and the resulting protocols.

In figure 2, we present a modification of the original fixed UB algorithm presented in [22] (underlined text shows the main differences). The modification corresponds to the *priority sequencing* class of algorithms and it is actually very similar to the original fixed UB algorithm. The main difference is that incoming messages are not immediately sequenced and sent to all the destinations but queued according to their priority and later sent.

```
Sender:
    Procedure TO-broadcast(m, prio):
        prio(m) := prio
        send m to sequencer
Sequencer:
    Initialisation:
        seqnum := 1
        incoming := {}
    Parallel: when receive (m):
        insert m in incoming, according to prio(m)
    Parallel: after initialisation:
        while incoming is not empty do
            m := first message in incoming
            incoming := incoming \ {m}
            sn(m) := seqnum
            send (m, sn(m)) to all
            seqnum := seqnum + 1
Destinations:
    Initialisation:
        nextdeliver := 1
        pending := ∅
    when receive (m, seqnum):
        pending := pending ∪ {(m, seqnum)}
        while ∃ (m, seqnum) ∈ pending : seqnum = nextdeliver do
            deliver m
            nextdeliver := nextdeliver + 1
```

Figure 2: Modification of fixed UB.

A modification of the privilege-based algorithm of [22] is shown in figure 3. This modification corresponds to the *priority sending* class of algorithms.

In figure 4, we show the modification of the causal history algorithm shown in [22]. This modification corresponds to the *priority delivering* class of algorithms. As the original algorithm, it assumes that a FIFO channel is available.

Finally, in figure 5 we show a destinations agreement algorithm that fits into the third subclass identified in [22] and corresponds to the *priority-based consensus* class of algorithms. This sketch is presented just

```
Sender (code of process $p$):
    Initialisation:
        tosend := $\underline{\{\}}$
        if $p = s_1$
            token.seqnum := 1
            send token to $s_1$
    Procedure TO-broadcast (m, **prio**):
        insert m in tosend **according to prio**
    when receive token:
        while tosend is not empty do
            m := first message in tosend
            send (m, token.seqnum) to destinations
            token.seqnum := token.seqnum + 1
            tosend := tosend \ {m}
        send token to $s_{i+1 mod n}$
Destinations (code of process $p$):
    Initialisation:
        nextdeliver := 1
        pending := $\emptyset$
    when receive (m, seqnum):
        pending := pending $\cup$ {(m, seqnum)}
        while $\exists$ (m, seqnum) $\in$ pending : seqnum = nextdeliver do
            deliver m
            nextdeliver := nextdeliver + 1
```

Figure 3: Modified privilege-based algorithm

as an illustrative example about how to apply this technique. According to this protocol, the messages received by the nodes are not directly delivered to the application but queued in a list. From time to time, a *starter* node decides to start a consensus round by sending to all the nodes a special START_CONSENSUS message. When a node receives such a message, sends a response that contains a set with the identifiers of the pending messages. When the *starter* receives a minimum number k of responses then it decides the common subset to all the received sets. The *starter* then decides a suggested total order of the selected messages, taking into account the priorities of the messages. This suggested order is sent to all the nodes (by means of an APPLY_ORDER message) so they can deliver the selected messages in the proper priority-based total order.

In the proposed algorithm, we use the *order* procedure to select the common subset of messages and to order them according to their priorities. The *vote* procedure is used to locally decide if the order proposed by the *starter* node is accepted. We are not addressing the voting mechanism, but nevertheless, keeping this procedure apart from the *order* procedure makes possible to use any consensus [43] mechanism.

8 Related Work

Unlike plain total order broadcast, priority-based total order broadcast has not been too much studied. To the best of our knowledge, few results have been presented. In this section, we comment the more interesting results we have found.

In [35] (an extension of [34]), a starvation-free priority-based total order protocol is presented. The protocol sits on top of an existing total order broadcast service so the protocol in all the processes receives the messages in the same order. It then locally and deterministically orders messages according to their priorities.

The protocol keeps a queue of incoming messages that is ordered according to the priorities of the messages. Messages with the same priority are queued according to their arrival order. Incoming messages are queued in their corresponding place.

```
Senders and destinations (code of process $p$):
    Initialisation:
        received := $\emptyset$
        delivered := $\emptyset$
        LC := $\{0, \ldots, 0\}$
    Procedure TO-broadcast(m, prio)
        LC[p] := LC[p] + 1
        ts(m) := LC[p]
        prio(m) = prio
        send FIFO (m, ts(m)) to all
    when receive (m, ts(m))
        LC[p] := max(LC[p], ts(m)) + 1
        if p $\neq$ sender(m)
            LC[sender(m)] := ts(m)
        received := received $\cup$ {m}
        deliverable := $\emptyset$
        for each message m in received $\setminus$ delivered do
            if ts(m) $\leq \min_{q \in \Pi}$ LC[q] then
                deliverable := deliverable $\cup$ m
        deliver all messages in deliverable,
        in increasing order of (ts(m), prio(m), sender(m))
        delivered := delivered $\cup$ deliverable
```

Figure 4: Modified causal-history algorithm

High priority messages are in the head of the queue and low priority are in the tail of the queue. To deliver messages according to their priority, messages are taken from the head of the queue and delivered to the application. To avoid starvation of low priority messages a periodic timer is used. Every time the timer expires, the reordering of incoming messages is temporarily paused. The protocol then forces the delivery of as much messages as possible, so low priority messages have a chance to be delivered to the application.

Such protocol is an extension of some existing total order protocol rather than a total order broadcast protocol itself and does not integrate the priority management in the total order protocol core. For this reason, it cannot be classified according to the taxonomy of [22].

In [39], another priority-based total order protocol is presented. This protocol guarantees that a message that has been received by all the processes will be delivered in the same order by all the processes, before any other message of a lower priority that has not been delivered by any process yet.

The protocol keeps a list of incoming messages that is ordered according to the priority of the messages. This list has a common suffix in all the processes. As some processes are faster than others, they deliver messages to the application faster than others, so the head of the list is, in general, different.

When a message is sent, all the processes receive it (unless they fail). When a process receives a message, it blocks part of the list of the incoming messages (the part that contains messages of a lower priority). The process then sends some information related to the blocked part of the list of incoming messages to a special process that acts as a coordinator. It also sends information about the last messages delivered to the application. The coordinator uses all this information to decide in which point of the list processes must insert the incoming message.

This protocol undergoes starvation of low priority messages if too many high priority messages are sent. According to the authors, this protocol is especially suitable "for state machine-like applications where the time to consume a message is far greater than the time involved in the communication rounds" so no starvation issues are expected to appear, no solution is given nor even the problem itself is pointed at.

This problem may be solved as suggested in previous sections. From time to time, regular prioritisation may be blocked, and lower priority messages are given a chance to be delivered. As previously said, this solution may cause priority inversion, as high priority messages may have to wait for low priority messages.

```
Senders:
    Procedure TO-bcast (m, prio):
        prio(m) := prio
        send m to all
Destinations:
    Initialisation:
        incoming := {}
        ids := {}
    when receive (m):
        add m to incoming
        add id(m) to ids
    when receive (START_CONSENSUS) from starter:
        send ids to starter
    when receive (orderedIds):
        vote := vote(orderedIds)
        send vote to starter
    when receive (APPLY_ORDER(orderedIds)):
        ids := ids \ orderedIds
        while orderedIds is not empty do
            id := first id in orderedIds
            orderedIds := orderedIds \ {id}
            m := message in incoming with id
            deliver m
            incoming := incoming \ {m}
```

```
Starter:
    Initialisation:
        incomingIds := $\emptyset$
        votes := {}
    when start consensus:
        send START_CONSENSUS to all
    when receive (ids):
        add ids to incomingIds
        if $|incomingIds| \geq k$
            orderAndPropose()
    when receive (vote v):
        add v to votes
        if there is a majority of positive votes
            send APPLY_ORDER(orderedIds) to all
Procedure orderAndPropose:
    orderedIds := order(incomingIds)
    send orderedIds to all
```

Figure 5: Modified destinations agreement (subclass 3)

Regarding the classification of [22], as the delivery history is used to decide the order of the messages (as well as the priorities of the incoming and existing messages), this protocol may be classified in the *deterministic merge* subclass of the *communication history* protocol class.

9 Discussion About the End-to-end Argument

The end-to-end argument [40, 32] is a design principle that can be applied to many different kinds of systems. According to this principle, functionality used by an application and often packed as a library or any other external form (e. g. as a *service* somehow offered by the operating system) is better placed in the application level. Several reasons are given in favour of the end-to-end argument and against the opposite *low-level* argument.

In our case, the end-to-end alternative means taking into account message priorities at the application level while the low-level alternative means considering the priorities at the group communication system level, as presented in section 6.

In the end-to-end alternative, the application tags its messages with a priority level (as in the proposed modifications in section 6). Each node may send its messages according to their priorities. The messages are broadcast and totally ordered by means of a regular group communication system, and delivered to the application (in every node) according to a total order.

As this order does not reflect any priority order, the application is in charge of reordering the incoming messages according to their priorities. This reordering must be done in such a way that the total order property of the sequence of delivered messages is kept.

The reordering can be done in a distributed manner, by using some additional message rounds, which increase the cost of the priority-based total ordering service.

The reordering can alternatively be done in a local manner, by means of a deterministic method. In a first alternative, time can be logically divided as in [35]. Some special control message, periodically sent by some *coordinator* node and totally ordered respect regular messages can be used to decide the duration of the *time parts*. Messages in each time period can be locally ordered (also guaranteeing the total order property).

A second alternative consists in forcing each node to wait for a predefined number of incoming messages and then deterministically reorder them according to their priorities. This way, the control messages used in the first alternative are avoided.

Such end-to-end approaches offer the advantage of being compatible with any total order broadcast protocol. In [40], another advantage of such application-level kind of solutions is argued. An application may have some information which is not accessible by a lower level and this information may allow the application to tune the operation of the service in order to get better performance numbers. In our particular case, there is no special information accessible only to the application[3] that could be used to get a better performance so, in our case, this argument is actually not a real advantage.

On the other hand, these application-level solutions have some disadvantages that must also be considered. First of all, both solutions impose a delay in the delivery of the incoming messages (until the next control message arrives or the expected number of messages are received) which depends on the duration of each *time part* or the number of required messages.

Furthermore, in case a node fails and later recovers it must ask another node for the messages it has missed. If the priority management is performed at the application level, the application is in charge of keeping a list of recent incoming messages in order to forward them to the recovering node. In the first solution, each node must save, at least, all the messages from the last control message. In the second case, each node must save all the messages received since the last reordering took place. In both cases, this need complicates the design and implementation of the application. Instead, a middleware-based alternative, like any of the modifications we propose in section 6, offers a simple and powerful solution that allows the application designer to focus on the relevant aspects of the application.

Besides that, these application-level solutions are not much reusable. If no special efforts are made to keep this priority-based reordering in a modular component, then the solutions are not reusable at all.

Moreover, the design and the implementation of the application is more complicated, as pointed above. Indeed, the application not only must reorder messages according to their priorities but also respect the total order property of the original sequence of incoming messages. Application designers need to worry about concerns that are not directly related to the application core.

So, unless application designers very carefully design these non-core aspects of the application, there is a risk that additional software bugs are put in. Instead, if this functionality is designed and implemented by specialised designers, this risk can be highly reduced.

For all these reasons, our conclusion about the end-to-end argument is that, in this particular case, is not worthy at all to move to an application-level layer the priority-based (re)ordering of incoming totally ordered messages.

10 Conclusions and Future Work

In this work we analyse the priority-based total order broadcast problem from a theoretical point of view. We start from a taxonomy of total order broadcast protocols [23] that classifies a number of protocols into five different classes and identify four different ways of adding priority management to the protocols.

Each modification is illustrated with an algorithm. The algorithms shown in Figs. 2, 3 and 4 come from three algorithms proposed in [23]. The algorithm shown in Fig. 5 is a new proposal we introduce to illustrate the *priority delivering* technique.

As the algorithms show, the modifications are quite simple, so they can be applied not only to the original protocols but also to other existing protocols, published by any other author, that might even be tuned to work under particular constraints or in a particular environment, or even to existing implementations, as long as they can be classified into any of the classes identified in [23].

This work is a first step for anyone interested in tackling the problem of providing a priority-based total order broadcast service. Our next step is to start an experimental phase in which we will prototype

[3]Keep in mind that we are not comparing the use of an application-level priority-based reordering of an existing total order service against a regular total order broadcast service but against a modified total order broadcast service that takes into account message priorities

the proposed algorithms. Then they will be tested in different scenarios with varying application patterns, measure their performance and compare the numbers. Those results may allow us to decide which protocol is the most suitable for each configuration.

We might then try to propose an architecture similar to that proposed in [30] to dynamically change the current priority-based total order broadcast protocol used by an application according to changes to the environment, to the application behavior or to other parameters.

References

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

[2] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.*, 17(3):513–560, 1992.

[3] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *19th Annual ACM International Symposium on Principles of Distributed Computing (PODC-19)*, pages 209–218, Portland, OR, USA, 2000.

[4] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000)*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[5] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: specifications. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, 40127 Bologna, Italy, Sept. 1996.

[6] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: specification and algorithms. Technical Report UBLCS-98-01, Department of Computer Science, University of Bologna, Bologna, Italy, April 1998.

[7] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[8] Ziv Bar-Joseph, Idit Keidar, Tal Anker, and Nancy Lynch. Qos preserving totally ordered multicast. In *4th International Conference on Principles of Distributed Systems (OPODIS)*, pages 143–162, 200.

[9] Ziv Bar-Joseph, Idit Keidar, and Nancy A. Lynch. Early-delivery dynamic atomic broadcast. In *16th International Conference on Distributed Computing (DISC'02)*, pages 1–16, London, UK, 2002. Springer-Verlag.

[10] Piotr Berman and Anupam A. Bharali. Quick atomic broadcast. In *7th International Workshop on Distributed Algorithms (WDAG'93)*, pages 189–203, London, UK, 1993. Springer-Verlag.

[11] Ken Birman and Robert van Renesse, editors. *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[12] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distibuted systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[14] M. Chen and K. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2(1):325–346, 1990.

[15] Gregory Chockler, Nabil Huleihel, and Danny Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing*, pages 237–246, New York, NY, USA, 1998. ACM Press.

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

[17] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118:158–179, 1995.

[18] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Real-Time Systems*, 2(3):195–212, 1990.

[19] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Recovery strategies for linear replication. *Lecture Notes in Computer Science*, 4330:710–723, 2006.

[20] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[21] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[22] Xavier Défago, André Schiper, and Péter Urbán. Totally ordered broadcast and multicast algorithms: taxonomy and survey. Technical Report IS-RR-2003-009, École Polytechnique Fédérale de Lausanne, Sept. 2003.

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

[24] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Department of Computer Science, University of Toronto; Department of Computer Science, Cornell University, 1994.

[25] Luis Irún-Briz, José Enrique Armendáriz-Iñigo, Hendrik Decker, José Ramón González de Mendívil, and Francesc Daniel Muñoz-Escoí. Replication Tools in the MADIS Middleware. In *Workshop on Design, Implementation, and Deployment of Database Replication (VLDB 2005)*, pages 25–32, Trondheim, Norway, August 2005.

[26] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the amoeba distributed operating system. In *11th International Conference on Distributed Computing Systems (ICDCS'91)*, pages 222–230, April 1991.

[27] Idit Keidar and Danny Dolev. *Dependable Network Computing*, chapter Totally ordered broadcast in the face of network partitions, pages 51–75. Kluwer Academic Publications, 1999.

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

[29] S. W. Luan and V. D. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):271–285, 1990.

[30] Emili Miedes, Mari-Carmen Bañuls, and Pablo Galdámez. Group communication protocol replacement for high availability and adaptiveness. To appear in LNCS (2007).

[31] Hugo Miranda, Alexandre Pinto, and Luis Rodrigues. Appia: a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, 2001.

[32] Tim Moors. A critical review of 'end-to-end arguments in system design'. In *IEEE International Conference on Communications*, pages 1214–1219, 2002.

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

[34] Akihito Nakamura and Makoto Takizawa. Priority-based total and semi-total ordering broadcast protocols. In *12th International Conference on Distributed Computing Systems (ICDCS 92)*, pages 178–185, June 1992.

[35] Akihito Nakamura and Makoto Takizawa. Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel network. In *2nd International Symposium on High Performance Distributed Computing*, pages 281–288, July 1993.

[36] Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science (Elsevier)*, 291(1):79–101, Jan. 2003.

[37] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, Jan. 1994.

[38] Luís Rodrigues, José Mocito, and Nuno Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *ACM Symposium on Applied Computing, Dijon, France, April 23-27, 2006*, pages 723–727, 2006.

[39] Luís Rodrigues, Paulo Veríssimo, and Antonio Casimiro. Priority-based totally ordered multicast. In *3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control (AARTC'95)*, Ostend, Belgium, May 1995. IFAC.

[40] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[41] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[42] Alan Tully and Santosh K. Shrivastava. Preventing state divergence in replicated distributed programs. In *9th Symposium on Reliable Distributed Systems*, pages 104–113, Oct. 1990.

[43] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.

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

[45] Pedro Vicente and Luís Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *21st Symposium on Reliable Distributed Systems, Osaka, Japan*, pages 92–101, October 2002.

[46] Yun Wang, Emmanuelle Anceaume, Francisco Brasileiro, Fabíola Greve, and Michel Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, August 2002.

[47] Yun Wang, Francisco Brasileiro, Emmanuelle Anceaume, Fabíola Greve, and Michel Hurfin. Avoiding priority inversion on the processing of requests by active replicated servers. In *International Confererence on Dependable Systems and Networks (DSN'01)*, pages 97–106. IEEE Computer Society, 2001.

[48] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*, pages 206–215, 2000.

[49] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.