Reducing Transaction Abort Rates with Prioritized Atomic Multicast 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-07/22

Reducing Transaction Abort Rates with Prioritized Atomic Multicast Protocols

Emili Miedes, Francesc D. Muñoz-Escoí

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

Technical Report TR-ITI-ITE-07/22

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

October, 2007

Abstract

Priority atomic multicast (i.e., total-order multicast) is a message delivery service that allows an application to prioritize the delivery of certain messages over others, while keeping the regular total order properties. Such a service can be used by applications of different types. As an example, such a priority-based message delivery service can be used by a replication middleware to reduce the abortion rate of the transactions being applied. In [9] we identified different priorization techniques and studied how to apply them modifying different classes of total order protocols to offer a priority-based total order delivery service. In this report we continue this work presenting an experimental study of these techniques. In this study, we compare three *classical* total order protocols against their corresponding prioritized versions. To this end we use a test application that broadcasts *prioritized* messages using these protocols and use some criteria to measure the effect of the priorization. We show that, under certain conditions, the use of prioritized protocols yields lower abort rates respect to the corresponding non-prioritized protocols.

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

1 Introduction

A group communication service (GCS) offers a set of services that are usually employed as building blocks to design and implement a distributed system. A common service of a GCS is an atomic multicast (i.e. total order multicast) message delivery service, which allows a user application to send messages to a set of destinations and get all of them delivered by each destination in the same order. Atomic multicast has been studied during almost thirty years and during that time a huge number of results has been produced [4, 6, 3, 5, 10, 1]. Atomic multicast services can also offer an additional property to allow the user applications to prioritize certain messages over others [15, 13, 11].

Such a service can be used in a scenario like the following. Consider an application that runs on top of a database replication middleware and is physically distributed among several sites. Such systems usually follow a *constant interaction* model [17], according to which, updates made by a transaction are broadcast in total order to all the database replicas at the end of the transaction, using a single message. The 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 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.

To this end, we can alter the order in which messages are delivered by the replicas. The middleware may assign different priority levels to different transactions according to some criteria. Messages sent in the scope of a given transaction may be tagged with the corresponding priority level, and a priority-based group communication protocol may be used to broadcast transaction messages. As the group communication protocol prioritizes some messages over other messages, the result is that the updates corresponding to some transactions are prioritized over others. This might be needed in order to reduce the abortion rate when semantic integrity constraints are considered. For instance, a bank might compel a given set of customers to have positive balances in their accounts. Those accounts with low balance should prioritize insertions against withdrawals. A possible solution consists in assigning to each operation (transaction) a priority directly proportional to the amount being added to the balance (so withdrawals get a negative priority value, i. e., the worst priority). So, if multiple concurrent transactions exist, they will be ordered in a way that maximizes their success probability.

This priority-based total order broadcast service can also be used in other scenarios. For instance, consider a distributed application used to monitor an environment using some kind of sensors. The sensors constantly generate and broadcast a number of messages (e.g. sensor readings) to a number of endpoints (e.g. human operators or other software or hardware subsystems). A prioritized total order broadcast may be used to prioritize the delivery of readings produced by critical sensors. Prioritized messages could also be used to signal exceptional values in a sensor reading, in order to allow the human operators to quickly react to unexpected changes in the environment observed.

This report is a continuation of the work started with a previous technical report [9]. In the report, we presented several contributions regarding priority-based group communication. First of all, we proposed four techniques to modify existing total order broadcast protocols to take into account message priorities. For each technique, the cost of introducing priority management was analyzed. We then showed how these techniques can be applied and identify which technique is the most suitable for each of the classes of total order broadcast protocols presented in [6]. Finally, we presented one modified algorithm for each technique.

In this report, we present some experimental results that show the benefits of applying the proposed techniques. The report is organized as follows. In Sect. 2 we show the system model we are assuming. In Sect. 3, we review the classification of total order broadcast protocols given in [6]. In Sect. 4 we briefly sketch several techniques to modify the different classes of total order protocols to take into account the priorities of the messages. We then show, in Sect. 5, some early experimental results of a comparison among original and modified protocols. In Sect. 6, we present a few references about priority-based total order, classify them according to the taxonomy of [6] and then compare them against the modifications proposed in Sect. 3. Finally, in Sect. 7 we conclude the report.

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*. 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.

Each process has a multilayer structure. In Fig. 1 we show the structure corresponding to the first example of Sect. 1. There is a user level 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.

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 multicast messages. If an external process (like in the second example of the introduction) needs to multicast a message to the group, it simply forwards the message to

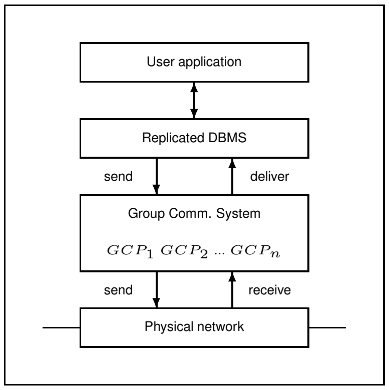

Figure 1: Multilayer architecture.

a group member that actually multicasts such message.

The system is partially synchronous. 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.

Processes can fail due to several reasons (for instance, hardware failures, software bugs or human misoperation). Processes are also subject to network failures that keep them from sending or receiving messages. Network partitions may also occur. Nevertheless, since we are focusing on the comparison of priorization 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 assume that lower values of $P(m)$ correspond to higher priorities and higher 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 Reviewing Atomic Protocols

In [6], 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 a *fixed sequencer* protocol, a single process is in charge of ordering the messages. In a *moving sequencer* protocol, sequencing is performed by a single process, as in a *fixed sequencer* protocol, but in this case 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 a *privilege based* protocol, processes can only send messages when they are allowed to do it. If just a process is allowed to send messages in every moment, then the total order can easily be set using just a global sequence number.

In a *communication history* protocol, processes use historical information about message sending, reception and delivery to totally order messages. In [6], two different types of *communication history* protocols are identified: *causal history* protocols and *deterministic merge* protocols. The class of *causal history* protocols is based on the total order mechanism proposed in [8]. 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 in 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. 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 a *destinations agreement* protocol, some kind of agreement protocol is run to decide the order of one or more messages. In [6], 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.

4 Priority Management

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 briefly sketched in the following sections. In [9], a detailed explanation of each technique can be found. We also analyze in [9] the cost that each technique imposes on the original protocol and provide pseudo-code sketches to show how these techniques can be applied.

4.1 Priority Sequencing

Priority sequencing may be applied to sequencer-based total ordering protocols like fixed sequencer or moving-sequencer protocols. The idea is to keep a list of unsequenced messages, ordered according to the priorities with which messages are tagged. The sequencer sequences each message in the order in which it appears in the list.

This scheme is quite simple but low priority messages may undergo a starvation problem [9].

4.2 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, some protocols of the *deterministic merge* subclass of the *communication history* protocol class and the first class of *destinations agreement* protocols, presented in [6].

The idea is to use a priority-ordered list of outgoing messages in each node and send them according to that order. Once sent, messages may be treated according to the final protocol used to totally order the messages.

4.3 Priority Delivering

The protocols of the *causal history* subclass of the *communication history* class of [6] 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, i.e., 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 ordering concurrent messages taking into account the priorities of the messages before 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.

4.4 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, presented in [6].

The modification, which is actually quite similar to that of Sect. 4.3, 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.

5 Experimental work

In this section we present some experimental work we have done in order to compare original and modified total order protocols. First of all we describe the environment and the application used to do the tests. Then, we describe the methodology we have followed to run the tests as well as the parameters used. Finally, we present some figures and discuss the results.

5.1 Environment

To test the original and the prioritized protocols, we have designed a simple application which will be described later. The application is ran under different configurations that will also be described later.

The experiments have been ran in a system of a single node with an Intel Pentium 4 processor at 2.8 GHz and 1 GB of RAM, running Linux Fedora Core Release 1. To compile and execute all the Java classes we used the Sun JDK 1.5.0.

The application uses the services of a total order protocol which in turn uses the services of a reliable transport layer. This layer is our own implementation of the sixth transport protocol presented in [14]. This reliable transport uses the services provided by an unreliable transport we built on top of the bare UDP sockets provided by the Java platform.

5.2 Test application

We have designed a test application that keeps track of the overall amount of money being managed by a stock trade company for all its stock investors. Each broker of the company runs its own instance (or *node*) of the application and operates on the stock exchange on behalf of the stock investors.

The application is composed of two main components. A first component continuously analyzes the stock market and suggests the most interesting options (stock selling or purchasing) according to different factors. A second component keeps track of the global balance of the stock trade company. When a broker decides to perform some operation suggested by the first component, the second component verifies the operation and applies the required updates to the global balance. If the operation implies the purchase of shares, the second component must check that it can be performed, considering the price of the purchase and the current global balance of the company. In some cases, this component rejects an operation (for instance, when the price of the purchase exceeds the global balance of the company).

As there are a number of brokers working or the company buying and selling shares, the global balance is continuously updated accordingly to the operations. In order to guarantee that the current value of the company's balance is consistent among all the nodes of the application, a total order protocol is needed. The total order protocol is used by all the nodes of the application to multicast the updates that are being applied so all the brokers see the same sequence of operations and apply the same sequence of updates to the global balance. This way, in every moment the global balance is consistent among all the nodes of the application.

We use a `BalanceTest` application to simulate the second component. The application is composed of a number of concurrent threads, each one representing a node managed by a different broker. Each node creates and broadcasts a number of messages, each one representing an operation (stock selling or buying) that may update the current balance. Each update carries an integer value. A positive value represents a stocks selling operation and the number is an increment to be applied to the global balance. A negative value represents a stocks purchasing operation and the number is a decrement on the global balance. To simplify the analysis of the results, we adopted the following convention regarding the range of the updates of the balance the application will allow. The integer values belong to a range whose upper limit is 1000. The lower value of the range can be parameterized, as we will show in Sect. 5.4. The value assigned to each message is computed by a random generator. Additional details about the seeds used in the tests will be given in Sect. 5.3.

All the messages are multicast to all the nodes using a total order protocol so all the messages are delivered by all the nodes in the same order. Nodes apply messages in the order they are delivered by the total order protocol. Applying a message means updating the global balance kept by the node. As all the nodes receive the same sequence of updates, all of them keep consistent their corresponding copy of the balance.

As described below, each node multicasts a sequence of messages, each one representing an update of the global balance corresponding to a stocks operation and its priority. This way we can simulate the normal operation of a regular stock trade company that has a number of brokers performing stock operations on behalf of several hundreds of investors.

A stock market is a very dynamic scenario in which a decision (for instance, to buy a number of shares by a given price) applied out of time can cause disastrous results. Due to the complexity of the stock markets, some decisions are more urgent than others, and, in some cases, must be prioritized over others. For this reason, a stock trading application needs some mechanism that allows the brokers to set the priority of the operations they are issuing, in order to get the highest benefits from them.

In our test application, each message also carries a second integer value that represents the priority of the message. The priority of a given message expresses the urgency of the corresponding operation. In a real stock trade application, these priorities are usually computed taking into account a big number of factors, like the status of the market, the recent evolution of the shares, some long-term histories, the risk of the operation (for stocks purchasing) or the expected benefit (for stocks selling), among many others.

In our test application we considered a simplified approach that easies the design and implementation of the application and also the analysis of the results. The priority is computed deterministically from only the value of the operation (purchasing or selling). Given the value v corresponding to an operation, its priority p is computed as $p = 1000 - v$. According to this expression, an update of the global balance with a value of 1000 has a priority of 0 and an update with a value of -1000 has a priority of 2000. Taking into account that priority management in the modified total order protocols is implemented considering a *the lowest value, the highest priority* rule, then the first update has a higher priority than the second one. In other words, we are prioritizing positive updates (from a share sale) over negative updates (from a share purchase).

In `BalanceTest`, we implemented another rule to *discard* some negative updates. When an update is about to be applied, the new balance is computed. If the new balance is greater or equal to zero, then the update is applied. If the new balance is negative, then the update is *discarded*. In other words, we do not allow the global balance to be in the red. This rule is used to show the differences between conventional (non-prioritized) and prioritized total order protocols.

5.3 Methodology

The expected behavior of an execution of `BalanceTest` depends on the conventional or prioritized type of the total order protocol used. When using a non-prioritized protocol, the nodes apply approximately the same number of positive and negative updates. When using a prioritized protocol, positive updates (sales) are prioritized, as stated before. This means that the balance kept by the nodes will be increased faster than decreased and less negative updates will be discarded.

To compare a prioritized total order protocol against the non-prioritized version of the same protocol, we run the `BalanceTest` application and count the number of updates that have been discarded in both versions. A number of messages discarded by `BalanceTest` using a prioritized protocol lower than the number when it uses the corresponding non-prioritized protocol means that the prioritized protocol has been able to prioritize a number of messages. The higher the difference is, the best is the priorization achieved by the prioritized protocol.

To test the proposed techniques, we tested different protocols. For each protocol, we varied the number of messages broadcast by each node. We also tried different values for the lower bound of the numeric value of the updates. For each combination of these parameters, we executed the `BalanceTest` and got the number of updates discarded. The concrete values of these parameters are discussed in Sect. 5.4.

As usual, a single execution of `BalanceTest` is not reliable enough to test a prioritized protocol against its non-prioritized version. To get reliable results, each execution is repeated a number of times. Each execution yields a number of discarded updates and we can compute the mean and median values of all the executions of a given test. We then compare the mean and median numbers of discarded updates by both versions of a given total order protocol.

Finally, we tried to avoid that the sequence of messages sent by each node had influence on the results. We forced each node to send the same sequence of messages (i.e. the same sequence of priorities) in each test ran with the same combination of the parameters and each protocol. This way, in the same test (combination of the rest of the parameters) all the protocols receive the same sequence of priorities, which allows us to notice better the differences among the protocols. As said before, each test is ran a number of times, but it is not necessary to send exactly the same sequence of messages (priorities) in each execution of the series. The only really needed is to guarantee that in the i-th execution of a given test (combination of the parameters), a given node sends the same sequence of messages with all the protocols.

5.4 Parameters

In this section we describe the values of the parameters used to test the protocols. First of all, we describe a set of *fixed* parameters, whose values are the same for all the tests. Then we explain a set of *variable* parameters.

Each test is ran by a single Java Virtual Machine that runs a single `BalanceTest` instance. This instance spreads four threads, representing four nodes. Each thread creates a sequence of messages, as described above, and sends them using a fixed sending rate (currently, 50 messages per second).

Each message is tagged with a priority value that ranges between a fixed maximum value equal to 1000 and a variable minimum value, described below.

The variable parameters are the protocol type, the number of messages sent by each node and the minimum bound for updates.

Regarding the protocols, we have implemented three non-prioritized total order protocols and then modified them to get the corresponding prioritized protocols. The *UB* protocol is an implementation of the UB sequenced-based total order algorithm proposed by [7][1]. The *UB_PRIO* protocol is the corresponding prioritized version of *UB*. The *TR* protocol implements a token ring-based algorithm, which is, in essence, similar to the ones of [10] and [1]. The *TR_PRIO* protocol is the corresponding prioritized version of *TR*. Finally, the *CH* protocol is an implementation of the causal history algorithm shown in [6].

We have executed different tests in which each node receives 400, 2000 and 4000 messages, respectively.

Moreover, we have used two different values for the minimum lower bound of the range for the integer values that represent the updates. The values used have been -1000 and -1200. Thus, the range for the updates are $[-1000, 1000]$ and $[-1200, 1000]$, respectively.

The combination of these values yields a total of 36 different settings. For each setting, we have ran 500 executions of the `BalanceTest` application.

5.5 Results

For each execution of the `BalanceTest` application we got the number of discarded messages. For each series of executions, we have got a series of 500 numbers of the discarded messages in each of those 500 executions. Then we got the mean and median values of each series and represented the medians graphically.

To represent the medians, we have divided them in two different groups, depending on the value used as a lower bound of the range for the integer values that represent the updates. In the first group, the medians correspond to a lower bound equal to -1000, while in the second group, the lower bound is equal to -1200. In Figures 2 and 3 we show the medians for the first and second group, respectively. In the X axis we

[1]UB stands for *Unicast-Broadcast*, as in [6].

represent the number of messages received by each node (400, 2000 and 4000 messages). In the Y axis we represent the number of discarded messages. The value represented is the median for each series of 500 executions.

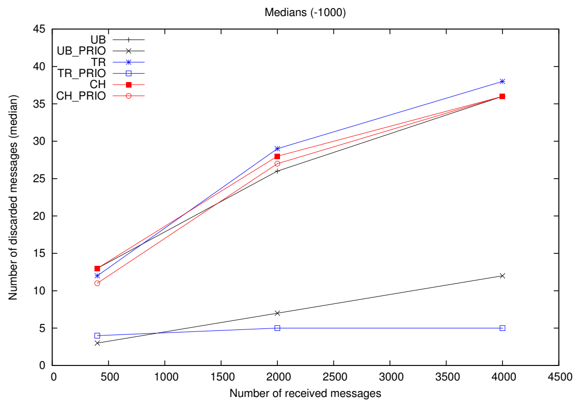

Figure 2: Median numbers (lower bound equal to -1000)

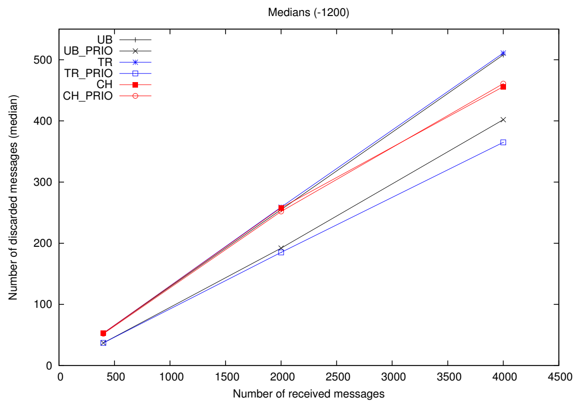

Figure 3: Median numbers (lower bound equal to -1200)

5.6 Discussion

The experiments show that the priorization techniques yield good results.

When the lower bound of the balance updates is -1000, the prioritized versions of the *UB* and *TR* protocols offer a very important reduction on the number of discarded messages, respect their original counterparts. Nevertheless, the reduction is lower in case of the *CH_PRIO* protocol respect to the original *CH* protocol. When the lower bound is -1200, the reduction is lower but still important in case of the *UB* and *TR* protocols.

In Table 1, we summarize the reduction we got in each case (in percentage values), considering different lower bounds and number of messages received per node.

As shown by Figures 2 and 3 and Table 1, the reduction offered by the *CH_PRIO* protocol is negligible. As explained in [9], the original *CH* protocol offers a message delivery service with total and causal order guarantees and the modified protocol must necessarily offer the same guarantees. This means that the modified protocol must ignore message priorities when reordering and delivering causally dependent messages and can only take into account message priorities to order and deliver concurrent (causally independent)

	-1000			-1200		
	400	1000	4000	400	1000	4000
UB_PRIO	76%	73%	66%	35%	24%	20%
TR_PRIO	66%	82%	86%	30%	28%	28%
CH_PRIO	15%	3%	0%	1%	2%	1%

Table 1: Percentage abort rate reduction

messages. As the number of causally independent messages is small, the priorization mechanism included in the *CH_PRIO* protocol can only offer a very small improvement respect to the original *CH* protocol.

Regardless the protocol, the lower bound of the balance updates also has a high influence on the results. When the lower bound is equal to -1000 the number of discarded messages is significantly lower, regardless the protocol used and the number of received messages per node, respect to the same setting ran with a lower bound equal to -1200.

In the second case, the interval is $[-1200, 1000]$, which means that negative values are more likely than positive ones and therefore withdrawals are more likely than deposits. As there are more withdrawals than deposits, the balance gets smaller and smaller and withdrawals have a higher probability to be discarded. Nevertheless, in the first case, the interval of possible values for a balance update (deposit or withdrawal) is $[-1000, 1000]$. Positive and negative values have the same likelihood and therefore the number of discarded messages is lower case than in the previous case.

There are other factors to consider when analyzing the effects of the prioritized protocols. The most determining factor is probably the application and the use it makes of the system. First of all, to benefit from a prioritized total order protocol, an application must send messages with different priorities. Moreover, prioritized protocols are only advantageous when there is a sustained flow of prioritized outgoing messages, sent at a minimum sending rate. If an application sends prioritized messages with a low sending rate, the prioritized messages are quickly handled, ordered and delivered, without colliding with other prioritized messages and no significant benefit is get from the prioritized protocols.

6 Related Work

Priority-based total order broadcast has not been studied so much as regular total order and, to the best of our knowledge, few results have been presented.

In [12] (an extension of [11]), 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. For this reason, it cannot be classified according to the taxonomy of [6].

In [13], 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.

Regarding the classification of [6], 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.

The protocol presented in [13] undergoes starvation of low priority messages if too many high priority messages are sent. The problem of message starvation is specifically addressed in [12].

In [2, 16] another common problem of this kind of protocols, known as *priority inversion*, is addressed.

7 Conclusions and Future Work

In this work we present an experimental study of different techniques to add support to prioritized messages. As discussed in [9], these techniques can be applied to different existing total order protocols.

In this work, we implemented several conventional (non-prioritized) total order protocols. We also implemented their corresponding prioritized versions, using the priorization techniques discussed in [9].

Then, we have tested the different protocols using a simple test application. Each update is tagged with a priority (which is actually ignored by non-prioritized total order protocols). These priorities are used to prioritize stocks sales over purchases. Updates can be discarded (not applied) according to some criteria (the global balance can not be in the red; see Sect. 5.3).

When the test application using a conventional non-prioritized total order protocol, a number of messages is usually discarded, due to randomness, according to the criteria chosen. When the test is repeated under the same conditions, but using the corresponding prioritized version of the same total order protocol, the number of discarded messages is usually lower (see Sect. 5.6). The difference between these two results can be used to measure the benefit using the prioritized version of a total order protocol (see Sect. 5.3) respect the original one.

The results got with the test application suggest that, under certain settings, an application can benefit from a prioritized total order protocol. Moreover, these early results also suggest a possible comparison among the prioritized techniques and protocols.

As described in Sect. 5.2, the test application is currently composed by a number of threads that simulate independent nodes. At first sight, it seems that the setting is not *real* enough in order to draw some conclusions from. Nevertheless, this setting allows to compare the protocols taking out even the effect of a real network.

The next step consists in testing the conventional and prioritized protocols in a more conventional setting, in which there is a real network. We plan to repeat the experiments in a cluster of commodity machines, connected by a conventional local area network. These experiments will allow us to evaluate the impact of actually using a network.

Additional tests are planned, in order to test other factors, especially those related to application-level usage patterns.

Finally, we pretend to propose an architecture to dynamically change the current priority-based total order broadcast protocol used by an application according to changes in its environment (overall system load, application current load), to the application behaviour or to other parameters.

References

[1] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Dependable Systems and Networks*, pages 327–336, Washington, DC, USA, 2000. IEEE-CS.

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

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

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

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

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

[7] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the amoeba group communication system. In *16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 436–448, Washington, DC, USA, 1996. IEEE Computer Society.

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

[9] Emili Miedes and Francesc D. Muñoz-Escoí. Adding Priorities to Total Order Broadcast Protocols. Technical Report TR-ITI-ITE-07/07, Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, March 2007.

[10] 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.

[11] Akihito Nakamura and Makoto Takizawa. Priority-based total and semi-total ordering broadcast protocols. In *12th Intl. Conf. on Dist. Comp. Sys. (ICDCS 92)*, pages 178–185, June 1992.

[12] Akihito Nakamura and Makoto Takizawa. Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel network. In *2nd Intl. Symp. on High Performance Dist. Comp.*, pages 281–288, July 1993.

[13] 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.

[14] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[15] 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.

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

[17] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Symposium on Reliable Distributed Systems*, pages 206–215, 2000.