Reducing Transaction Abort Rates with Prioritized Atomic Multicast Protocols*

Emili Miedes, Francesc D. Muñoz-Escoí, and Hendrik Decker

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Campus de Vera s/n, 46022 Valencia (Spain)
{emiedes,fmunyoz,hendrik}@iti.upv.es

Abstract. Priority atomic multicast is a total-order multicast message delivery service that enables applications to prioritize the sequence by which messages are delivered, while regular total order properties remain invariant. Priority-based message delivery can serve to reduce the abortion rate of transactions. In this study, we compare three classical total order protocols against their corresponding prioritized versions, in the framework of a replication middleware. To this end, we use a test application that broadcasts prioritized messages by these protocols, and measure the effect of the priorization. We show that, under certain conditions, the use of prioritized protocols yields lower abort rates than the corresponding non-prioritized protocols.

1 Introduction

A group communication service (GCS) is a software package that provides a set of building blocks for designing and implementing distributed systems. Atomic (i.e., total-order) multicast message delivery is a standard GCS building block, which enables an application to send messages to a set of destinations such that they are delivered in the same order by each destination. Atomic multicast has been studied for more than thirty years, during which a huge amount of results has been produced [1,2,3,4,5,6]. Some of these services offer an additional feature that enables human or programmed users and agents to prioritize the delivery of certain messages over others [7,8,9].

Such a service can be used in a scenario like the following. Consider an application that runs on top of a database that is transparently replicated over several sites by means of a middleware. Such database systems usually behave according to a *constant interaction* model [10], by which the updates of a transaction are broadcast at its end in total order to all replicas using a single message. The order in which a set of messages are delivered by the replicas determines the final sequence in which those transactions are applied to the database. Different sequences may have different execution properties, such as performance, resource

* This work has been partially supported by EU FEDER and the Spanish MEC under grant TIN2006-14738-C02-01 and by IMPIVA under grant IMIDIC/2007/68.

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 394–403, 2008.

consumption and likelyhood of abortion. In particular, if improperly or unfortunately prioritized, transactions may violate some semantic integrity constraints, and thus must be aborted, but otherwise, they can commit.

A simple example is given by two concurrent transactions which increment and, respectively, decrement a value constrained by some upper and lower thresholds: clearly, prioritization of one transaction over the other may lead to a constraint violation and therefore abortion of the transaction processed first, while the reverse prioritization will cause no such problems.

In general, the idea investigated in this paper is to alter the order in which transactions are committed for achieving a favorable constraint evaluation, thus reducing their abort rate. To this end, the middleware may assign different priorities to different transactions according to some (possibly application-dependent) criteria. Messages sent in the scope of a given transaction can be tagged with the corresponding priority, and a priority-based group communication protocol can be used to broadcast transaction messages. As the protocol prioritizes some messages over others, the respective updates will be prioritized correspondingly. As indicated above, this may be beneficial for reducing the number of abortions due to integrity constraint violations.

As another example, consider a distributed application for controlling critical and non-critical remote systems. Messages sent to critical systems can be prioritized over messages sent to non-critical ones, by means of a priority-based total order protocol.

Non-prioritizing total broadcast policies have been widely studied, while, as far as we know, only a few studies exist for priority-based protocol variants. In [11] (an extension of [9]), a starvation-free priority-based total order protocol is presented. In [8], another priority-based total order protocol is presented. Low priority messages may suffer starvation if too many high priority messages are sent. The problem of message starvation is dealt with specifically in [11]. In [12,13] another common problem of this kind of protocols, known as *priority inversion*, is addressed.

This paper is a follow-up to [14], in which we proposed four ways to extend existing total order broadcast protocols (as classified in [2]), by priority assignment to messages. In this paper we present the results of applying the proposed techniques to an application the transactions of which are subject to semantic integrity constraints. Messages involve requests to modify data that would cause semantic constraints to become violated, so that the modifications are rejected. We show how message prioritization can be used to increase the likelyhood of satisfying the semantic constraints of an application.

In Sect. 2 we describe the assumed system model. In Sect. 3 we review the classification of total order broadcast protocols in [2]. In Sect. 4 we briefly sketch several techniques to modify such protocols in order to prioritize messages. Sect. 5 discusses how priorization support can be integrated in a database replication middleware. In Sect. 6, some experimental results of comparing original and modified protocols are exposed. In Sect. 7 we conclude with an outlook to future investigations.

2 System Model

In this section we briefly recapitulate the system model assumed throughout the paper. A complete description can be found in [14].

The considered system is composed of a set of processes that communicate through message passing. Each process has a multilayer structure, whose topmost level is a user application that accesses a replicated DBMS, which in turn uses the services offered by a group communication system (GCS). The latter is composed of one or more group communication protocols (GCP), which use the underlying network's services to send and deliver messages.

The system is partially synchronous. We assume that processes run on different physical nodes and the drift between two different processors is not known. Moreover, the time needed to transmit a message from one node to another can be bounded.

Processes can fail due to several reasons. Also network partitions may occur. However, since we are focusing on the comparison of priorization techniques, we are not going to address failure handling (which can be realized by mechanisms such as group membership services and fault-tolerance protocols).

3 Reviewing Atomic Protocols

A survey of total order protocols is given in [2], where total order protocols are partitioned into five classes.

In a *fixed sequencer* protocol, a single process is in charge of ordering the messages. In a *moving sequencer* protocol, sequencing is also performed by a single agent, but its role is transferred from one process to another.

In a *privilege based* protocol, processes may only send messages when they have permission to do so. If a process has permission to send messages at any time, then the total order can easily be set using a global sequence number.

In a *communication history* protocol, processes use historical information about message sending, reception and delivery to totally order messages. In [2], two different types of *communication history* protocols are identified: *causal history* protocols and *deterministic merge* protocols.

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

4 Priority Management

In [14], we have identified four basic techniques for adding priority management to total order protocols, including detailed explanations, a cost analysis and pseudo-code outlines. Essentially, they differ mostly with regard to the point in the messages' life-cycle in which priorities are considered.

Priority Sequencing. It can be applied to sequencer-based total ordering protocols like fixed- or moving-sequencer protocols. The idea is to maintain a list of yet unsequenced messages, ordered according to their priority tags. The sequencer then sequences each message in the order of that list. This scheme is quite simple, but low priority messages may suffer starvation [14].

Priority Sending. It 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, as presented in [2]. 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 can finally be treated according to the protocol used to totally order the messages.

Priority Delivering. This technique can be applied to the *causal history* subclass of the *communication history* class of [2]. It consists in ordering concurrent messages (i.e., those that are not causally dependent on each other), 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.

Priority-based Consensus. It is applicable to the second and third classes of *destinations agreement* protocols, presented in [2]. The modification, which is actually quite similar to that of the priority delivering technique, 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. Nevertheless, both ordering and priorization rely on a consensus among all nodes. This poses additional problems when failures happen. Moreover, these protocols are quite sensitive to delays occurring in a single node, which may end up delaying the operation of the whole system. Therefore, we decided to ban this technique and the corresponding classes of protocols from the experiments outlined in Sect. 6.

5 Integration in Database Replication Systems

Priority assignment mechanisms as outlined in Sect. 4 can be automated modularly in the framework of a replication middleware such as MADIS [15]. Such a module obtains information about declarative constraints (and, in some DBMSs, also about triggers) from the DBMS catalog tables; e.g., in the pg_constraint table of PostgreSQL 8.x. From those, it infers which kind of updates may violate any constraint. For instance, the application described in Sect. 6 depends on CHECK constraints. Such constraints can be found in the table mentioned above with all information (i.e., the constraint expression and the affected columns) needed for adequately prioritizing transactions. In general, the set of all update operations susceptible to integrity violation can be stored in some meta-table, and ordered according to some priority heuristics. The priority module may then scan the operations of each transaction and assign the tabled priority to it.

Instead of preconfiguring priority assignments statically, they may be determined dynamically, by some external component that communicates with our middleware, once the transaction has been started. To this end, specialized operations should be added to the middleware API.

6 Experimental Work

In this section we present some experimental work we have done in order to compare original and modified total order protocols. First, we describe the testbed, then, the parameters and the methodology used to run the tests. Finally, we present and discuss the results.

6.1 Environment

The application uses the services of a total order protocol which in turn uses a reliable transport layer. This layer is our own implementation of the sixth transport protocol presented in [16]. It is based on the services provided by an unreliable transport we built on top of the bare UDP sockets provided by the Java platform.

The experiments have been conducted in a system of four nodes with an Intel Pentium D 925 processor at 3.0 GHz and 2 GB of RAM, running Debian GNU/Linux 4.0 and Sun JDK 1.5.0. The nodes are connected by means of a 22-port 100/1000Mbps DLINK DGS-11224T switch that keeps the nodes isolated from any other node, so no other network traffic can influence the results.

6.2 Test Application

Our test application keeps track of the overall amount of money being processed by all investment brokers of a stock trade enterprise. Each broker runs its own instance (or *node*) of the application, operating on the stock exchange on behalf of the stock owners and a potentially large number of investors.

When a broker decides to perform some operation, the application attempts to apply the requested updates to the global balance. If the operation implies the purchase of shares, the application must check that it can be performed, considering the price of the purchase and the current global balance of the enterprise. The application rejects an operation when the price of the purchase exceeds the global balance.

As there are several brokers working at various sites for the company buying and selling shares concurrently, the global balance is incessantly updated. In order to ensure that the current value of the global balance is consistent among all nodes of the application, a total order protocol is needed. It is used by all nodes to multicast the updates so that all brokers see the same sequence of operations and apply the same sequence of updates to the global balance. That way, consistency among all nodes at each moment is achieved.

Each node creates and broadcasts a number of messages, each one representing a stock trading operation that may update the current balance. Each update carries an integer value. Positive and negative values represent selling and buying

operations of stock trading, respectively. To simplify the analysis of the results, we adopted the following convention. The integer values range from -1000 to 1000. The actual value assigned to each message is generated at random.

All messages are multicast to all nodes using a total order protocol, so all messages are delivered by all nodes in the same order. Nodes apply messages in the order as received from the total order protocol. To apply a message means to update the local copy of the global balance, as kept by each node. Since all nodes receive the same update sequence, their corresponding copies of the balance are kept consistent.

Each message carries a second integer value which represents its priority. In real-life stock trading, these priorities are determined by considering a large number of factors, such as the market situation, recent evolutions of shares, some long-term trends, risk analyses, expected benefits, etc.

The priority of each operation is uniquely determined by its type (purchase or sale), as follows. Given the value v of an operation, its priority p is computed as $p = 1000 - v$. Thus, a sale update of the global balance with a value of 1000 obtains the priority value 0, and a purchase update with a value of -1000 obtains priority 2000. Since priority management in the modified total order protocols is implemented according to a *lower value = higher priority* rule, the priority of the first update is higher than that of the second one. So, positive updates (from sales) are prioritized over negative updates (from purchases).

In `BalanceTest`, we implemented the constraint for discarding updates that would overdraw the balance. For each negative update request, the presumptive new balance is computed. If it is greater or equal to zero, then the update is applied. Otherwise, the update is discarded. Thus, the global balance is prevented from ever being in the red. This constraint serves to highlight the benefits that can be obtained using prioritized instead of conventional, non-prioritized total order protocols.

6.3 Methodology

The expected behavior of an execution of `BalanceTest` is different for the conventional and the prioritized protocol versions. For the former, the nodes apply approximately the same number of positive (sale) and negative (purchase) updates. For the latter (prioritized) version, positive updates (i.e., sales transactions) are prioritized, as already stated. This means that the balance is more likely to increase than to decrease, thus less purchase transactions will be discarded.

To test the proposed prioritization techniques, we tested different protocols. For each protocol, we varied two parameters: the number of messages broadcast by each node, and the numeric value of the update request. The values of these parameters and their combination for different test runs are detailed in Sect. 6.4. For each parameter combination, we executed `BalanceTest` and recorded the number of discarded updates.

For obtaining reliable results, each execution of `BalanceTest` has been repeated a statistically relevant number of times. Each execution resulted in a

number of discarded updates, so that we could compute the mean and median values of all executions of each test combination. For each total order protocol, the final result was then constituted by comparing the amount of discarded updates of the prioritized and the non-prioritized version.

We took care that the measured results were independent of the sequence of messages sent by each node. We achieved that by having each node send the same sequence of messages, thus obtaining the same distribution of priorities in each test run, with the same parameter combination for each protocol. As said before, each test was run a statistically significant number n of times, but not necessarily with exactly the same sequence of messages (priorities) in each execution series. We have made sure, though, that each node sends the same sequence of messages for each protocol, each parameter vector and each iteration of the test series.

6.4 Parameters

In this section, we describe the values of the test parameters. First, we describe a group of *fixed* parameters, whose values are the same for all tests, and then a group of *variable* parameters.

Each `BalanceTest` instance is run in a physical node. Each instance creates a sequence of messages, as described above, and sends them by a fixed rate (currently, 50 messages per second). Each message is tagged with a priority value ranging between -1000 and 1000. The initial balance value is set to 0.

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

We have implemented three non-prioritized total order protocols and a prioritized version for each. The *UB* protocol is an implementation of the UB sequenced-based total order algorithm proposed by [17][1]. The *TR* protocol implements a token ring-based algorithm: in essence, it is similar to the ones of [5] and [6]. Finally, the *CH* protocol is an implementation of the causal history algorithm in [2]. The corresponding prioritized versions are *UB_PRIO*, *TR_PRIO* and *CH_PRIO*.

We have executed different tests in which each node receives 400, 2000 and 4000 messages, respectively. For each setting, we have ran 500 executions of the `BalanceTest` application.

6.5 Results

For each execution of `BalanceTest`, we recorded the number of discarded messages. Thus, for each of the 500 series of executions, we obtained a number of discarded messages. Then we calculated the mean and median values of each series and displayed the medians graphically.

In Fig. 1a) we represent the medians of the result series. The number of discarded messages is displayed along the Y axis, as a function of the number of messages received by each node (400, 2000 and 4000), on the X axis. The displayed value is the median for each series of 500 executions.

[1] UB stands for *Unicast-Broadcast*, as in [2].

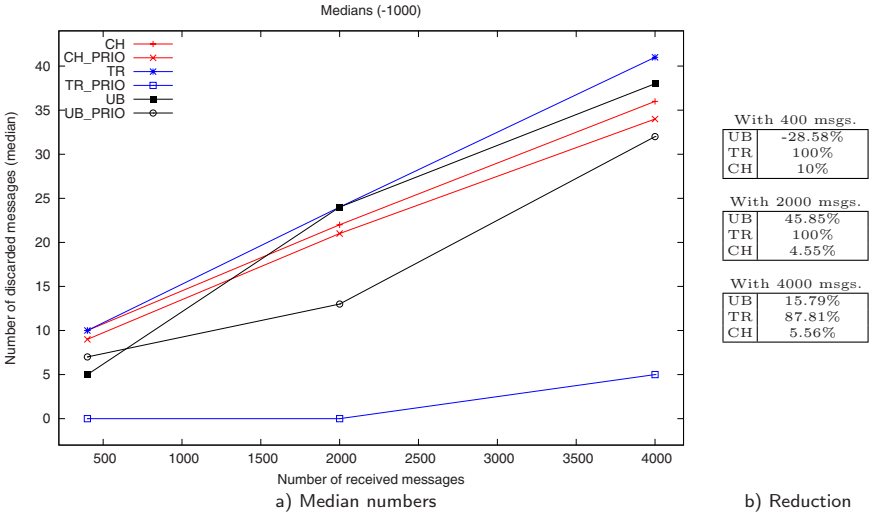

Fig. 1. Discarded messages

6.6 Discussion

The experiments show that the priorization techniques yield good results. The prioritized versions of the *UB* and *TR* protocols offer a significant reduction on the number of discarded messages, with regard to their original counterparts. For the *CH_PRIO* protocol, however, the reduction is lower, with regard to the original *CH* protocol.

Fig. 1b) summarizes the percentages of reduction obtained for each pair of protocols, with different lower bounds of numbers of received messages.

As shown in Fig. 1, the reduction chalked up by the *CH_PRIO* protocol is negligible. This is because the modified protocol must ignore message priorities when reordering and delivering causally dependent messages, as explained in [14]. It can only take into account message priorities for concurrent (causally independent) messages. As the number of causally independent messages is small, the priorization mechanism in the *CH_PRIO* protocol achieves only a very small improvement of the original *CH* protocol.

In [18] we also include the results for a similar set of tests ran with an update lower bound equal to -1200 and show that the lower bound of the balance updates influences the results considerably. When equal to -1000, the number of discarded messages is significantly lower, regardless of the protocol used and the number of messages received per node, when compared to runs with a lower bound of -1200, for the same setting.

When the interval is *[-1200, 1000]*, negative values are more likely than positive ones, so withdrawals are more likely than deposits. Thus, the balance keeps diminishing, so that withdrawals are discarded with increasing likelyhood. When

the interval is *[-1000, 1000]*, positive and negative values are equally likely, which is why the number of discarded messages is lower than for the previous interval.

There are other factors to be considered before drawing final conclusions from analyzing the effects of prioritized protocols. The most ponderous factor is probably the application. First of all, an application must send messages by a conscious choice of different priority tags if it wants to benefit from a prioritized total order protocol. Moreover, prioritized protocols are advantageous only if there is a sustained flow of prioritized outgoing messages, sent at a minimum and high sending rate. In conclusion, this means that the benefits of prioritization are highly application-dependent.

7 Conclusions

We have presented an experimental study of different techniques for supporting prioritized messages. We implemented several conventional (non-prioritized) total order protocols and their corresponding prioritized versions, using the techniques discussed in [14] and tested them with a simple application in which a semantic constraint is defined. The results show that the application benefits from using prioritized versions of total order protocols.

Currently existing group communication systems do not include priorization support in their total order protocols. The present results confirm that applications can benefit by making clever use of priority options offered by augmented versions of standard total order protocols. It is likely that existing group communication systems may be improved by adding, to their total order protocols, some priorization support based on our techniques. Our main contribution is to have shown that prioritized total order protocols are beneficial for applications that prioritize their transactions by taking into account their likelyhood of violating given constraints.

As a by-product, the testbed also provides a point of departure for a systematic study of more general comparisons, both between standard total order protocols and prioritization techniques.

A greater challenge for future investigations, however, is given by the goal to design, develop and implement an integrity checking mechanism that automatically assigns priorities to concurrent transactions in order to lower the rate of integrity violations and thus to lower the rate of abortions that are due to integrity violations. For achieving this goal, note that our test scenario is conceptually independent of the question whether integrity is checked by a built-in DBMS module or by some user-defined mechanism. An application-independent solution could be provided by an architecture disposing of a number of parallel processors for checking the preservation of integrity by various alternatives of sequentializable schedules, preferring those that result in less integrity violations and thus in a lower number of aborted transactions.

References

1. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computing Surveys 33(4), 427–469 (2001)
2. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys 36(4), 372–421 (2004)
3. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems 5(1), 47–76 (1987)
4. Dolev, D., Malki, D.: The Transis approach to high availability cluster communication. Communications of the ACM 39(4), 64–70 (1996)
5. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R., Lingley-Papadopoulos, C.: Totem: a fault-tolerant multicast group communication system. Comm. of the ACM 39(4), 54–63 (1996)
6. Amir, Y., Danilov, C., Stanton, J.R.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: DSN, pp. 327–336 (2000)
7. Tully, A., Shrivastava, S.K.: Preventing state divergence in replicated distributed programs. In: 9th Symposium on Reliable Distributed Systems, pp. 104–113 (1990)
8. Rodrigues, L., Veríssimo, P., Casimiro, A.: Priority-based totally ordered multicast. In: 3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control (1995)
9. Nakamura, A., Takizawa, M.: Priority-based total and semi-total ordering broadcast protocols. In: 12th Intl. Conf. on Dist. Comp. Sys (ICDCS 1992), pp. 178–185 (1992)
10. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: SRDS, pp. 206–215 (2000)
11. Nakamura, A., Takizawa, M.: Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel network. In: 2nd Intl. Symp. on High Performance Dist. Comp., pp. 281–288 (1993)
12. Baker, T.: Stack-based scheduling of real-time processes. Journal of Real-Time Systems 3(1), 67–99 (1991)
13. Wang, Y., Brasileiro, F., Anceaume, E., Greve, F., Hurfin, M.: Avoiding priority inversion on the processing of requests by active replicated servers. In: Dependable Systems and Networks, pp. 97–106. IEEE Computer Society, Los Alamitos (2001)
14. Miedes, E., Muñoz-Escoí, F.D.: Managing priorities in atomic multicast protocols. In: ARES: Intl. Conf. on Availability, Reliability and Security (2008)
15. Irún-Briz, L., Decker, H., de Juan-Marín, R., Castro-Company, F., Armendáriz-Íñigo, J.E., Muñoz-Escoí, F.D.: MADIS: A slim middleware for database replication. In: Euro-Par Conf., August 2005, pp. 349–359 (2005)
16. Tanenbaum, A.S.: Computer Networks. Prentice Hall, Englewood Cliffs (1996)
17. Kaashoek, M.F., Tanenbaum, A.S.: An evaluation of the Amoeba group communication system. In: 16th ICDCS, pp. 436–448. IEEE-CS, Los Alamitos (1996)
18. Miedes, E., Muñoz-Escoí, F.D.: Reducing transaction abort rates with prioritized atomic multicast protocols. Technical Report TR-ITI-ITE-07/22, Instituto Tecnológico de Informática, Universidad Politécnica de Valencia (October 2007)