A Fast Delivery Protocol for Total Order Broadcasting

Li Ou*, Xubin He*, Christian Engelmann†‡, and Stephen L. Scott†
*Department of Electrical and Computer Engineering
Tennessee Technological University, Cookeville, TN 38505, USA
Email: {lou21, hexb}@tntech.edu
†Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
Email: {engelmannc,scottsl}@ornl.gov
‡Department of Computer Science
The University of Reading, Reading, RG6 6AH, UK

Abstract—**Sequencer, privilege-based, and communication history algorithms are popular approaches to implement total ordering, where communication history algorithms are most suitable for parallel computing systems, because they provide best performance under heavy work load. Unfortunately, post-transmission delay of communication history algorithms is most apparent when a system is idle. In this paper, we propose a fast delivery protocol to reduce the latency of message ordering. The protocol optimizes the total ordering process by waiting for messages only from a subset of the machines in the group, and by fast acknowledging messages on behalf of other machines. Our test results indicate that the fast delivery protocol is suitable for both idle and heavy load systems, while reducing the latency of message ordering.**

I. Introduction

Total order broadcasting is essential for group communication services [1]–[3], but the agreement on a total order usually bears a cost of performance: a message is not delivered immediately after being received, until all the communication machines reach agreement on a single total order of delivery. Generally, the cost is measured as latency of totally ordered messages, from the point the message is ready to be sent, to the time it is delivered by the sender machine.

Traditionally three approaches are widely used to implement total ordering: sequencer, privilege-based, and communication history algorithms [3]. In sequencer algorithms, one machine is responsible for ordering the messages on behalf of other machines in the group. Privilege-based algorithms rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. For example, in a token-based algorithm [4], a token is rotated among machines in the same group, and one machine can only send messages while it holds the token. In communication history algorithms, total order messages can be sent by any machine at any time, without prior enforced order, and total order is ensured by delaying the delivery of messages, until enough information of communication history has been gathered from other machines.

Three types of algorithms have both advantages and disadvantages. Sequencer algorithms and privilege-based algorithms provide good performance when a system is relatively idle. However, when multiple machines are active and constantly send messages, the latency is limited by the time to circulate the token or produce the order number from the sequencer. Communication history algorithms have a post-transmission delay [3], [5]. To collect enough information, the algorithm has to wait for a message from each machine in the group, and then deliver the set of messages that do not causally follow any other, in a predefined order, for example, by sender ID. The length of the delay is set by the slowest machine to respond with a message. The post-transmission delay is most apparent when the system is relatively idle, and when waiting for response from all other machines in the group. In the worst case, the delay may be equal to the interval of heart beat messages from an idle machine. On the contrary, if all machines produce messages and the communication in the group is heavy, the regular messages continuously form a total order, and the algorithm provides the potential for low latency of total order message delivery.

In a parallel computing system, multiple concurrent requests are expected to arrive simultaneously. A communication history algorithm is preferred to order requests among multiple machines, since such algorithm performs well under heavy communication loads with concurrent requests. However, for relatively light load scenarios, the post-transmission delay is high. We propose a *fast delivery* protocol to reduce this post-transmission delay. The *fast delivery* protocol forms the total order by waiting for messages only from a subset of the machines in the group, and by fast acknowledging a message if necessary, thus it fast delivers total order messages. To verify our protocol, we implemented a prototype in the Transis [5] group communication system and applied this protocol to an active/active replication scenario.

This research was partially sponsored by the Office of Advanced Scientific Computing Research of the U.S. Department of Energy. The work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725. The work performed at Tennessee Tech University was partially supported by the U.S. National Science Foundation under under Grant Nos CNS-0617528 and SCI-0453438. This work was also partially sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory.

Let m_i and m_j are two total order broadcasting messages:

1) if m_i causally precedes m_j, all machines that deliver m_i and m_j deliver m_i before m_j.
2) if m_i and m_j are concurrent, if machine p delivers m_i before m_j, then any machine q that belongs to the same partition with p delivers m_i before m_j.

Fig. 1. The definition of total order broadcasting service.

II. Total Order Broadcasting Service

In this section, we briefly discuss the services a total order broadcasting system should provide. We assume that there is a substrate layer providing basic broadcasting services.

A. Basic Broadcasting Services

The machines in the system use a broadcasting service to send messages. A broadcast message is sent once by it's source machine, and arrives to all destined machines in the system, at different time. The broadcasting service is responsible for the reliable delivery of messages. Internally, the *causal* delivery order [6] of messages is guaranteed by the service.

(Definition): Message m_i and message m_j are *concurrent*, if m_i does not causally precedes m_j, and m_j does not causally precedes m_i.

The basic broadcasting service receives the messages of the network. It keeps causal order of messages and delivers them to our fast delivery protocol. The broadcasting service does not guarantee the same delivery sequence of concurrent messages on all machines in the system.

B. Total Order Broadcasting

On top of the basic broadcasting service, totally ordered broadcasting extends the underlying causal order to a total order for concurrent messages.

(Total Order): If two correct machines p and q both deliver message m_i and m_j, then p delivers m_i before m_j if and only if q delivers m_i before m_j.

The total order broadcasting provided by the system does not guarantee the total order across multiple partitions. As long as partitions do not occur, all machines deliver the messages in the same total order. When a partition occurs, machines in every partition continue to form the same total order. However, this total order may differ across partitions. The total order broadcasting service of the system is defined in Fig. 1.

III. Fast Delivery Protocol for Total Order Broadcasting

In this section, we describe a *fast delivery* protocol to provide the total order broadcasting service defined in Section II-B. We assume that the protocol works on top of the basic broadcasting services described in Section II-A. We first consider a static system of n machines, which means no failure of machines, no network partitions and re-merges, no new machines. Those features will be considered in the Section IV, in which we show how to extend the protocol to handle dynamic environments.

Each machine will not deliver any messages until it collects a message set from other machines in the group. The message set should contain enough information to guarantee the totally ordered delivery. After receives enough messages, the machine delivers the set of messages that do not causally follow any other, in a predefined order. Idle machines periodically broadcast heart beat messages with a predefined interval on behalf of other machines. Those heart beat messages will not be delivered, but used by machines to determine the order of received messages.

A. Notation and Definition

We define that a partition P consists of a group of machines $\{p_1, p_2, \cdots, p_N\}$. We assume each machine in the group P has a distinct ID. For a machine p, function $id(p)$ returns its ID. If the number of machines in the primary group P is N,

$$\begin{aligned} &\forall p, q \in P, \quad id(p), id(q) \in \{1, 2, \cdots, N\}, \\ &id(p) \neq id(q) \end{aligned}$$

We associate with each machine $p \in P$ the functions $prefix$ and $suffix$ which are defined:

1) $prefix(p) = \{q | \forall q \in P, id(q) < id(p)\}$
2) $suffix(p) = \{q | \forall q \in P, id(q) > id(p)\}$

The input to the *fast delivery* protocol is a stream of causally ordered messages from underlying the broadcasting service. We denote by $m_{q,i}$ the ith message sent by machine q. $sender(m_{q,i}) = q$. If message $m_{q,i}$ is delivered before $m_{k,j}$ in machine p, $deliver(m_{q,i}) < deliver(m_{k,j})$

We define a *pending message* [7] to be a message that was received by the protocol but has not be agreed to total order, thus, not delivered for processing. A *pending message* that follows only delivered messages is called a *candidate message*. The set of concurrent candidate messages is called the *candidate set*. This is the set of messages that are considered for the next slot in the total order. For example, a system has 5 machines, $\{p_1, p_2, p_3, p_4, p_5\}$. After a certain time, there is no undelivered messages on any machines. Machine p_1 broadcasts a message m_p1, and machine p_4 broadcasts a message m_p4. All five machines receive both m_p1 and m_p4, but none of them can deliver the two messages, because except the sending machines, no one knows if messages m_p2 and m_p3 are sent by p_2 and p_3, concurrently with m_p4, or not. All machines should not deliver m_p1 and m_p4, until enough information is collected to determine a total order. The message set of m_p1 and m_p4 is called the candidate set, and messages m_p1 and m_p4 are called candidate messages. Let $M_p = \{m_1, m_2, \cdots, m_k\}$ be the set of candidate messages in a machine p. We associate with M_p a function $senders$.

$$senders(M_p) = \{sender(m_i) | \forall m_i \in M_p\}$$

Let Md_p is the set of messages ready to be delivered in a machine p. $Md_p \subseteq M_p$. The Md_p is called the *deliver set*.

```
When receiving a regular message $m$ in machine $p$:
  1) if ($m$ is a new candidate message)
         add $m$ into candidate set $M_p$
  2) if ($M_p \neq \phi$)
         for all ($m_i \in M_p$)
           if $prefix(sender(m_i)) \subseteq senders(M_p)$
                 add $m_i$ into delivery set $Md_p$
  3) if ($Md_p \neq \phi$)
         deliver all messages $m_j$ in $Md_p$ in the
         order of $id(sender(m_j))$
  4) if ($sender(m) \notin suffix(p)$)
         return
     if (message $m$ is not a total order message)
         return
     if (messages are waiting to be broadcast from $p$)
         return
     if ($\exists m_i \in M_p, id(sender(m_i)) = p$)
         return
     otherwise, fast acknowledge $m$
```

Fig. 2. The fast delivery protocol.

B. The Fast Delivery Protocol

The *fast delivery* protocol is symmetric, and we describe it for a specific machine p (the pseudo code is shown in Fig. 2). The key idea of the *fast delivery* protocol is that it forms the total order by waiting for messages only from a subset of the machines in the group. Assuming a candidate message m is in *candidate set* M_p, we use the following delivery criterion to define what messages a machine has to wait before delivering m:

1) Add m into *deliver set* Md_p when:

$$prefix(sender(m)) \subseteq senders(M_p)$$

2) Deliver the messages in the Md_p with the following order:

$$\begin{aligned} &\forall m_i, m_j \in Md_p, \\ &id(sender(m_i)) < id(sender(m_j)) \\ &\longrightarrow deliver(m_i) < deliver(m_j) \end{aligned}$$

With the same example in Section III-A, we explain how the protocol works. All five machines could deliver m_p1 immediately, because $prefix(p1) = \phi$, and $senders(M_p) = \phi$. The five machines could not deliver m_p4, because $prefix(p4) = \{p_1, p_2, p_3\}$, and $senders(M_p) = p_1$. Machines have to wait messages from both p_2 and p_4, but do not need to wait a message from p_5, because $p_5 \notin prefix(p4)$.

According to the protocol, if $prefix(sender(m)) \not\subseteq senders(M_p)$. the machine p has to wait messages from other machines before delivering m. If any of those machines is idle, the waiting time could be the interval of heart beat messages. To speedup the delivery of m, the idle machines should immediately acknowledge m on behalf of other machines. If a machine q receives a message m, and q is idle, q broadcasts a *fast acknowledgment* when:

$$sender(m) \in suffix(q)$$

In the same example, if p_2 and p_3 are idle, they should fast acknowledge m_p4, because $p_4 \in suffix(p_2)$, and $p_4 \in suffix(p_3)$. If p_5 is idle, it does not need to send fast acknowledgment because $p_4 \notin suffix(p_5)$.

fast acknowledgment reduces the latency of message delivery, however, it injects more packets into network. If communication is heavy, fast acknowledgment may burden network and machines, thus increase delivery latency. To reduce the cost of fast acknowledgment, we define the following acknowledgment criterion:

(ACK) Fast acknowledge a message m from a machine q when:

1) Message m is a total order message.
2) There is no message waiting to be sent from the machine q.
3) $\nexists m_j \in M_p, id(sender(m_j)) = q$.

It is very straightforward of conditions 1. Condition 2 means if a machine is sending regular messages, it is not a idle machine, and the regular messages themselves are enough to form a total order. Condition 3 means if a machine already sent a regular message which is still in the M_p, that message can be used to form a total order, without an additional acknowledgment. In the same example, if p_1 is idle after sent m_p1, it does not need to send any acknowledgment (although $p_4 \in suffix(p_1)$), because m_p1 is still in M_p.

In a parallel system, when multiple concurrent requests arrive a machine simultaneously and the system is busy, conditions 2 and 3 are very unlikely to be satisfied simultaneously, so almost no additional acknowledgments are injected into the network when communication is heavy.

IV. Fast Delivery Protocol for Dynamic Systems

The fast delivery protocol operates on an asynchronous stream of causal order messages. In this section, we show how to extend the protocol to handle failures, the network partitioning and re-merge, and joining machines.

Fast delivery protocol is integrated into the group communication service to provide total order delivery of messages on top of the basic broadcasting service. We assume that the system contains membership service, which maintains a view of *current membership set* (CMS) consistent among all machines in the dynamic environment. When machines crash or disconnect, the network partitions and re-merges, or the new machines join, the membership service of all connected machines must reconfigure and reach a new agreement of CMS.

After a new agreement is reached, membership service delivers a *view_change* event indicating a new configuration. All connected machines in the new configuration agree on the set of regular messages that belong to the previous membership, and must be delivered before the new view_change event. Fast

# of machines	Single	All
1	235	
2	952	971
3	1031	1461
4	1089	1845
5	1158	2236
6	1213	2646
7	1290	3073
8	1348	3514

TABLE I
REQUEST LATENCY (μs) COMPARISON OF MULTIPLE MACHINES. ”SINGLE” MEANS ONLY ONE MACHINE IN A RELATIVELY IDLE SYSTEM BROADCASTS REQUESTS. ”ALL” MEANS ALL MACHINES IN A BUSY SYSTEM BROADCAST REQUESTS.

delivery protocol is extended to define how to deliver such messages in the dynamic environment.

We assume that after a new agreement of CMS, the membership service notifies fast delivery protocol with a special event. With such event, the protocol gets the machine set, P_f, which belongs to previous configuration, but is included in the new configuration. The new $prefix(p)$ and $suffix(p)$ are calculated based on the P_f:

1) $prefix(p)_{new} = prefix(p)_{old} - P_f$
2) $suffix(p)new = suffix(p)_{old} - P_f$

Using the algorithm described in Section III, the set of regular messages that belong to the previous configuration are delivered before the view_change event with the new $prefix(p)$ and $suffix(p)$. Since a new CMS always completes within a finite delay of time, any total order message could be delivered within a limited time interval.

V. EXPERIMENTAL RESULTS

To verify above model, a proof-of-concept prototype for fast delivery protocol has been implemented in Transis [5], and deployed on the XTORC cluster at Oak Ridge National Laboratory, using up to 8 machines in various combinations for functional and performance testing. The computing nodes of the XTORC cluster are IBM IntelliStation M Pro series servers. Individual nodes contain a Intel Pentium 2GHz processor with 768MB memory, and a 40GB hard disk. All nodes are connected via Fast Ethernet (100MBit/s full duplex) switches. Fedora Core 5 has been installed as the operating system. Transis v1.03 with *fast delivery* protocol is used to provide group communication services. Failures are simulated by unplugging network cables and by forcibly shutting down individual processes.

An MPI-based benchmark is used to send concurrent requests from multiple machines. The latency is measured with blocked requests, and an average latency is calculated from 100 requests of each machine. The results under various configurations are provided for comparison from 1 to 8 machines (Table I). In the configuration of only one machine, the latency overhead mainly comes from processing cost of the group communication service. When the number of machines increases, additional overhead is introduced by the network communication and total order communication algorithm to reach agreement among machines. For each configuration, we measure the latency under both idle and busy systems. In an idle system, only a single machine is active to send requests. In a busy system, all machines send concurrent requests, and the communication traffic is heavy. The proof-of-concept prototype showed that although the latency increases with the number of machines, the *fast delivery* protocol works well to keep the overall overhead acceptable for any HPC system. The overhead is consistent for both idle and busy systems. The fast acknowledgment aggressively acknowledges total order messages to reduce the latency of idle systems when only a single machine is active. The protocol is smart enough to hold its acknowledgments when the network communication is heavy because more machines are involved.

We compared the fast delivery protocol with the traditional communication history algorithm provided by the original Transis system. In idle system, the post-transmission delay of traditional communication history algorithm is apparent. It is not a constant value, and in worst case, the latency is equal to the interval of heart beat messages from a idle machine. A typical interval is in the gratitude of hundred milliseconds, which is unacceptable, compared to the latency of fast delivery protocol. In a busy system, the latency of fast delivery protocol is almost the same as the traditional communication history algorithm, because the protocol holds unnecessary acknowledgments. We found that when all machines sent concurrent requests, the fast delivery protocol did not acknowledge any broadcast, and the regular messages continuously form a total order.

VI. RELATED WORK

Group communication systems typically provide different group broadcast services with a variety of ordering and reliability guarantees [1]. A totally ordered service extends the causal service by ensuring that messages sent to a set of processes are delivered by all these processes in the same order. Past research on total ordering algorithms focuses on three popular approaches [3]. Sequencer algorithms [8]–[10] rely on one machine to order the messages on behalf of other machines in the group. Privilege-based algorithms [4], [11] force the total order in the process of competition of a privilege among all machines in the group. Communication history algorithms [5], [7], [12]–[14] ensure the total order by delaying the delivery of messages, until the enough information of communication history has been gathered from other machines. The agreement on a total order in communication history algorithms usually bears a cost of performance: post-transmission delay [3], [5]. Several studies have been made to reduce the cost. Early delivery algorithms [7], [13] reduce the latency by reaching agreement with a subset of the machines in the group. Optimal delivery algorithms [15], [16] deliver messages before the total order is determined, but notify the applications and cancel the delivery if the final order is different with delivered order.

Researchers have considered using totally ordered broadcasting to provide high availability for HPC systems [17].

Past research in high availability primarily focused on the active/standby model [18]–[21]. Recent research of symmetric active/active replication model [22], [23] uses multiple redundant service nodes running in virtual synchrony [24] and totally ordered broadcasting.

VII. Conclusion

In this paper, we propose a fast delivery protocol to reduce the latency of message ordering in group communication. The protocol optimizes the total ordering process by waiting for messages only from a subset of the machines in the group. The protocol performances well for both idle and busy systems. Furthermore, the fast acknowledgment aggressively acknowledges total order messages to reduce the latency when some machines are idle. The protocol is smart enough to hold the acknowledgments when the network communication is heavy.

We implemented a prototype of the fast delivery protocol and discussed an application of this protocol in symmetric active/active replication in parallel computing systems. Our performance results are encouraging: they indicate that the fast delivery protocol is suitable for both idle and heavy loaded systems.

References

[1] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, 2001.

[2] R. Baldoni, S. Cimmino, and C. Marchetti, "Total order communications: A practical analysis," *Lecture Notes in Computer Science: Dependable Computing*, vol. 3463, pp. 38–54, 2005.

[3] X. Defago, A. Schiper, and Peter Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, December 2004.

[4] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Trans. Comput. Syst.*, November 1995.

[5] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Communications of the ACM*, 1996.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM*, July 1978.

[7] D. Dolev, S. Kramer, and D. Malki, "Early delivery totally ordered multicast in asynchronous environments," *Symposium on Fault-Tolerant Computing*, 1993.

[8] K. P. Birman and R. van Renesse, "Reliable distributed computing with the ISIS toolkit," *IEEE Computer Society Press*, 1993.

[9] M. F. Kaashoek and A. S. Tanenbum, "An evaluation of the Amoeba group communication system," *Proc. 16th Intl. Conf. on Distributed Computing Systems*, 1996.

[10] M. K. Reiter, "Distributing trust with the Rampart toolkit," *Communications of the ACM. 39*, vol. 39, no. 4, April 1996.

[11] F. Cristian, S. Mishra, and G. Alvarez, "High-performance asynchronous atomic broadcast," *Distributed System Engineering Journal 4, 2*, June 1997.

[12] M. K. Aguilera and R. E. Strom, "Efficient atomic broadcast using deterministic merge," *Proc. 19th Annual ACM Intl. Symp. on Principles of Distributed Computing*, 2000.

[13] Z. Bar-Joseph, I. Keidar, and N. Lynch, "Early-delivery dynamic atomic broadcast," *Proc. 16th Intl. Symp. on Distributed Computing*, 2002.

[14] D. Dolev and I. Keidar, "Totally ordered broadcast in the face of network partition," *Dependable Network Computing, Chapter 3*.

[15] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. Knowl. Data Eng.*, July 2003.

[16] P. Vicente and L. Rodrigues, "An indulgent uniform total order algorithm with optimistic delivery," *Proc. 21st IEEE Intl. Symp. on Reliable Distributed Systems*, 2002.

[17] C. Engelmann and S. Scott, "Concepts for high availability in scientific high-end computing," in *Proc. of HAPCW*, 2005.

[18] "PBSPro job management system for the cray XT3 at Altair Engineering, Inc," *http://www.altair.com/pdf/PBSPro Cray.pdf.*

[19] "SLURM at lawrence livermore national laboratory, Livermore, CA, USA," *http://www.llnl.gov/linux/slurm.*

[20] C. Engelmann, S. L. Scott, D. E. Bernholdt, N. R. Gottumukkala, C. Leangsuksun, J. Varma, C. Wang, F. Mueller, A. G. Shet, and P. Sadayappan, "MOLAR: Adaptive runtime support for high-end computing operating and runtime systems," *ACM SIGOPS Operating Systems Review*, 2006.

[21] I. Haddad, C. Leangsuksun, and S. L. Scott, "HA-OSCAR: Towards highly available linux clusters," *Linux World Magazine*, March 2004.

[22] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, "Active/active replication for highly available HPC system services," *Proceedings of the International Symposium on Frontiers in Availability, Reliability and Security (FARES),Vienna, Austria*, April 2006.

[23] K. Uhlemann, C. Engelmann, and S. L. Scott, "JOSHUA: Symmetric active/active replication for highly available HPC job and resource management," in *Proceedings of IEEE International Conference on Cluster Computing*, September 2006.

[24] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal, "Extended virtual synchrony," *Proc. of DCS*, 1994.