Ordered Multicast and Distributed Swap

Maurice Herlihy Srikanta Tirthapura
and Roger Wattenhofer

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-00-08
December 2000

Ordered Multicast and Distributed Swap

preliminary report

Maurice Herlihy* Srikanta Tirthapura*
Roger Wattenhofer†

August 13, 2000

Abstract

A multicast protocol is *ordered* (or *totally ordered*) if it ensures that messages multicast to a group of nodes are delivered in the same order at each destination node, even when those messages are generated concurrently from several sources. Ordered multicast is a natural foundation for push-based cache coherence and certain kinds of middleware.

This paper shows how to reduce the complex problem of enforcing multicast ordering to a simpler distributed coordination problem we call *distributed swap*. Any distributed swap protocol can transform an unordered reliable multicast into an ordered multicast in a modular way.

We introduce two novel distributed swap protocols, and discuss their corresponding ordered multicast protocols. These protocols have lower latency than more obvious approaches based on distributed counting.

1 Introduction

The *ordered multicast problem* is the problem of ensuring that messages multicast to a group of nodes are delivered in the same order everywhere, even when these messages are generated concurrently at several sources. (This

*Computer Science Department, Brown University, Providence, RI 02912; {herlihy, snt}@cs.brown.edu.

†Microsoft Research, rogerwa@microsoft.com

kind of multicast is sometimes called *totally ordered.*) We are interested in *scalable* multicast protocols that allow nodes to enter or leave the multicast group without the need for global reconfiguration.

To understand why ordered multicast can be useful, consider a distributed object cached at multiple nodes in a network, where each node multicasts updates to the cached copies, a technique called *push-based* cache coherence. If several nodes issue concurrent updates, each of the copies must apply those updates in the same order. More generally, ordered multicast is useful in any kind of middleware where the order of events originating at different nodes in a distributed system must be reflected consistently among the nodes "listening" to such events. For example, publish-subscribe systems such as Gryphon [3] can use such a service to provide ordered delivery of events to subscribers.

The contribution of this paper is to point out a simple reduction from the ordered multicast problem to a much simpler distributed coordination problem, called *distributed swap*, and to explore the consequences of this reduction. Our intent is to draw the attention of the community to this approach, and to encourage others to work on this problem and related problems. We have found this reduction helpful in designing two novel ordered multicast protocols, the *arrow multicast* and the *combining multicast*, both described below.

The reduction works as follows. A distributed swap protocol can be combined with any reliable multicast protocol from the literature (such as SRM [6] or RMTP [10]), to yield a reliable ordered multicast protocol. This reduction is of interest because distributed swap has not been widely studied, and there is some evidence that ordered multicast protocols based on distributed swap may be more efficient than more obvious approaches based on distributed counting.

We believe that any truly scalable multicast protocol must be *anonymous*, in the sense that a node performing a multicast may not be aware of the identities of the recipients. IP multicast [13], SRM [6], and RMTP [10] are all anonymous in this sense. If all nodes (or even some nodes) must know every group member, then entering or leaving the group requires a global reconfiguration, which is clearly not scalable beyond local area networks. Our approach to ordered multicast thus differs in fundamental ways from that employed by systems based on notions of *virtual synchrony* [1, 4, 12, 15], in which each node knows the exact group membership. In these systems, each node entering or leaving the group provokes a global reconfiguration (called a "view change"). These systems trade fault-tolerance for scalability, while we do the opposite.

2 Model

We now describe our assumptions about the underlying system and the services it provides. Formally, a distributed system is a connected undirected graph G, where processors correspond to graph nodes, and edges to direct communication links. We assume that there is

- a reliable FIFO unicast service (such as TCP), and
- a reliable single source FIFO[1] (but otherwise unordered) multicast service (such as SRM [6] or RMTP [10]).

Our goal is to provide a layer on top of these services that forces a total order on message delivery.

We believe that a layered approach, in which we focus exclusively on ordering, reflects a sensible separation of concerns. Many existing proposals for reliable multicast [6, 10, 11] stack a repair layer on top of an unreliable multicast layer. In the same spirit, we stack an ordering layer on top of reliable multicast so we can focus on ordering independently of repair and retransmission.

We distinguish between *receiving* a message at a node, which typically involves placing the message in a buffer, and *delivering* the message to the application running at the node. Ordered multicast requires only that message deliveries be totally ordered.

We do not consider fault tolerance for now. We assume here that nodes do not crash, and that with sufficient retransmission, any message can eventually cross any link. Fault tolerance is discussed briefly in the conclusions.

3 The Reduction

In the *distributed swap* problem, a collection of processes have to implement a distributed swap object, which encapsulates a value, initially the distinguished value $\perp$. The object provides a single operation, $swap(v)$, which changes the object's value to v and returns the object's previous value.

It is straightforward to use distributed swap to impose a total order on reliable multicasts. Each multicast group has an associated swap object. Each message m has a unique identifier, $id(m)$. Node u then calls $swap(id(m))$, which returns $id(m')$, the identifier of m', the immediate predecessor to m.

[1] "Single-source FIFO" means that messages sent by any *single* processor are delivered in the order sent.

Node u then calls the reliable multicast service to send the pair $\langle id(m'), m\rangle$ to each member of the multicast group. A node that receives $\langle id(m'), m\rangle$ delivers m only after delivering m'. (A message with predecessor $\perp$ is delivered immediately.)

To see why it may be advantageous to use swap to implement ordered multicast, let us consider another, perhaps more obvious reduction. The *distributed counting* problem requires issuing successive integers to requesting processes. One could implement ordered multicast simply by using any distributed counting protocol [2, 14, 16] to assign a *sequence number* to each message, and delivering a message only after all lower-numbered messages have been delivered.

Nevertheless, *it may sometimes be faster to identify a message's immediate predecessor than to discover the number of predecessors of that message.* Both distributed swap protocols examined in this paper are actually "truncated" versions of counting protocols, in which each participant quits early, after discovering its immediate predecessor, but without waiting to count its other predecessors. The task of finding a predecessor has lower latency in swap-based protocols than in the corresponding counting-based ones.

We present two techniques for ordered multicast based on distinct distributed swap protocols. The first protocol, the *arrow multicast*, is based on the *arrow distributed directory protocol* [5, 9]. The arrow protocol was originally developed as a scalable way for tracking the location of mobile objects. It uses a simple path-reversal technique to construct a distributed queue of access requests. The corresponding distributed swap protocol is a "truncated" version of the directory protocol, in which a message is multicast as soon as it chooses a position in the distributed queue.

The second protocol, the *combining multicast*, is based on a combining tree structure. The combining multicast protocol is a "truncated" version of a distributed counting protocol. In the basic combining-tree protocol, messages move from leaves to the root, combining wherever possible. In the counting protocol, each message is assigned a value only after a round trip to the root and back. The swap protocol is a truncated version of the counting protocol: when two messages are combined, one becomes the other's predecessor and the later one can be multicast immediately.

4 Arrow Multicast

The *arrow directory protocol* [5] ensures mutually exclusive access to mobile objects. This protocol is designed to avoid scalability problems inherent in many directory services currently used in distributed shared object systems. We now show how to adapt this protocol to support distributed swap, and hence ordered multicast.

Recall that the network is modeled as a graph G. The protocol maintains a tree T, a subgraph of G, spanning the nodes in the multicast group. Each node v in the tree has two attributes: $link(v)$ is a node, either v itself, or a neighbor of v in T, and $value(v)$ is a value. A node v is a *sink* if $link(v) = v$. The component $value(v)$ is meaningful only if v is a sink. The directory tree is initialized so that following the the $link(\cdot)$ variables from any node leads to a unique sink v such that $value(v) = \perp$. Informally, except for the unique sink node, a node knows only in which "direction" the sink lies. Figure 1 shows the initial system state..

When a node v initiates $swap(a)$, it sends a $swap(a)$ message to $u_1 = link(v)$ and sets $link(v)$ to v and $value(v)$ to a. When node u_i receives a $swap(a)$ message from node u_{i-1}, where $u_{i+1} = link(u_i)$, it immediately "flips" $link(u_i)$ to u_{i-1}. If $u_{i+1} \neq u_i$, then u_i forwards the message to u_{i+1}. Otherwise, u_i is a sink, and it returns $value(u_i)$ to the originating node v, and v's swap operation is complete.

Ordered multicast is performed in a similar way. When a node u multicasts m, it initiates $swap(id(m))$, sending m along with the swap message. When this message reaches a sink u_i, instead of returning $value(u_i)$ to u, u_i immediately performs an unordered multicast of $\langle value(u_i), m\rangle$, indicating that the message with id $value(u_i)$ is the immediate predecessor of m in the total order. This protocol is illustrated in Figures 1 through 5.

It is not hard to show that every message eventually finds a sink, and that the maximal number of links traversed from source to sink is at most the diameter of the tree T. Moreover, the protocol never waits between the time a message is multicast and the time it is received at the destination nodes. (Once a message m is received, however, a node may need to wait for earlier messages to catch up before it can deliver m.)

Note that this swap protocol can be considered as a truncated version of a distributed counting protocol. Interpret $value(v)$ as a mobile sequence number, initially 0. The arrow protocol forms a distributed queue of access requests (in this case requests for sequence numbers). The sequence number goes through the requests in the queue in order. Each request stores the current value of the sequence number, increments it and sends it on to the next request in the queue. It then performs an unordered multicast with sequence number that it has stored.

If requests occur sequentially, the counting-based and swap-based multicast protocols are the same, but in the presence of concurrency, the swap-

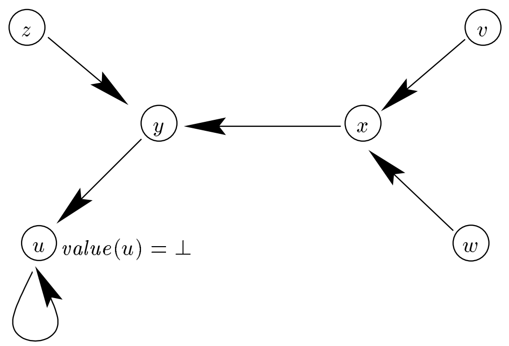

Figure 1: Arrow multicast. The initial system state.

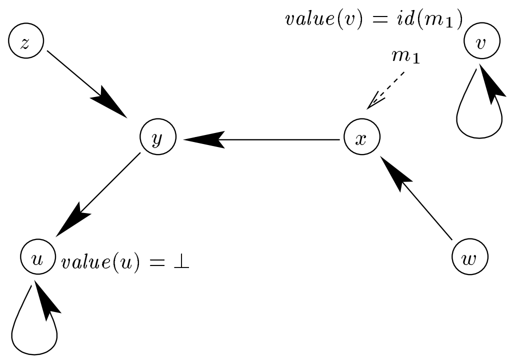

Figure 2: Arrow multicast. v sends message m_1, which is on its way to x.

based multicast will always have lower latency because messages do not need to wait to count their predecessors.

5 The Combining Multicast

Combining trees [7, 8] are a well-known technique for implementing distributed data structures in which concurrent requests are combined as they move up towards the root of a tree. We now show how a distributed counting protocol based on combining trees can also be truncated to yield a distributed swap protocol.

The directory maintains a tree T, a subgraph of G, spanning the members of the multicast group. The tree includes a *fixed root* r, and edges of the tree are oriented toward r. The root stores a value $value(r)$, initially $\perp$.

To execute $swap(a)$, where a is a value, a node v sends a $swap(a, a)$ message to its parent in T. If two messages $swap(a, b)$ and $swap(c, d)$ meet

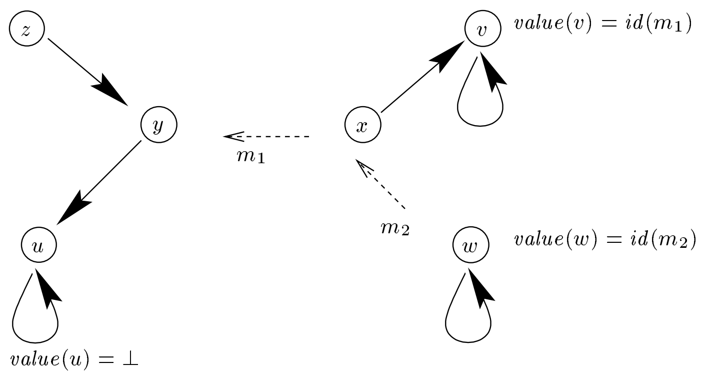

Figure 3: Arrow multicast. w sends m_2, now on its way to x.

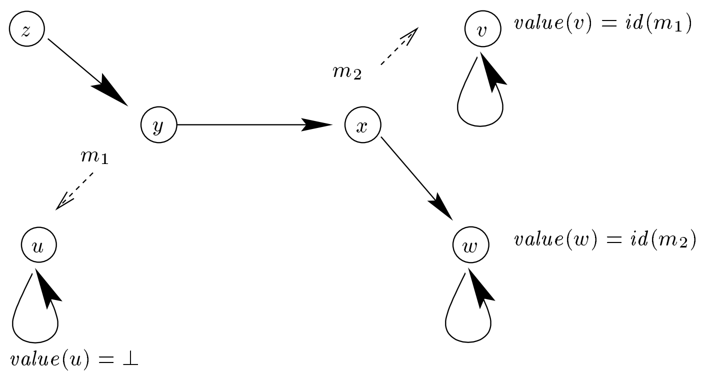

Figure 4: Arrow multicast. m_1 and m_2 follow the arrows, flipping their direction along the way. Note that m_2 has been "deflected" towards v.

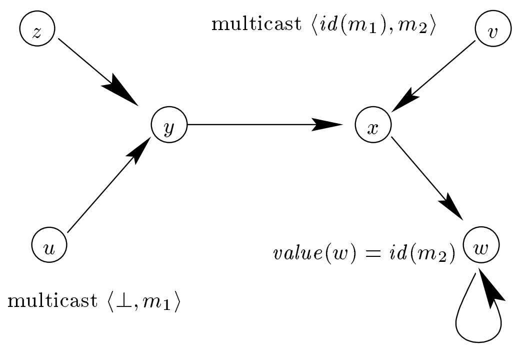

Figure 5: Arrow multicast. Both m_1 and m_2 find their predecessors concurrently and are multicast along with that information. Note that $value(v)$ and $value(u)$ are no longer useful.

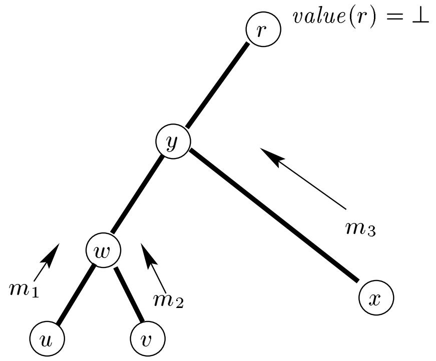

Figure 6: Combining multicast. Messages m_1, m_2 and m_3 start their journey towards the root.

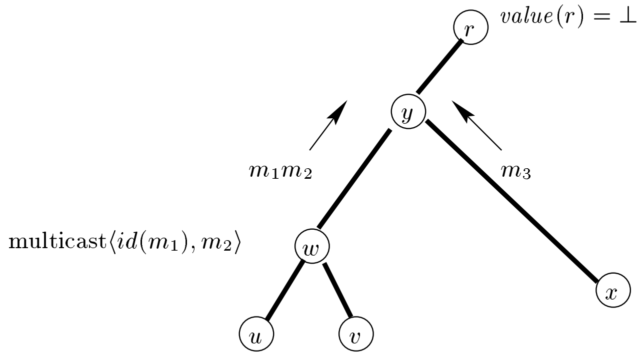

Figure 7: Combining multicast. m_1 and m_2 combine. m_2 is multicast as m_1's successor.

at a non-root node y, b is designated to be the result of c's swap, the messages are combined into a single message $swap(a, d)$, and the combined message is sent to y's parent. (More than 2 messages can be combined in a similar way.) When a $swap(a, b)$ message arrives at the root, the root's value $value(r)$ is designated to be the result of a's swap, and $value(r)$ is set to b.

An ordered multicast is performed in the same way, except that message identifiers replace values, and a node performs an unordered multicast as soon as it identifies a message's predecessor.

This swap protocol can also be viewed as a truncated counting protocol. If we were to use the combining tree to generate sequence numbers, then a message would be assigned a sequence number only after it had completed a round trip to the root (possibly combined with other messages).

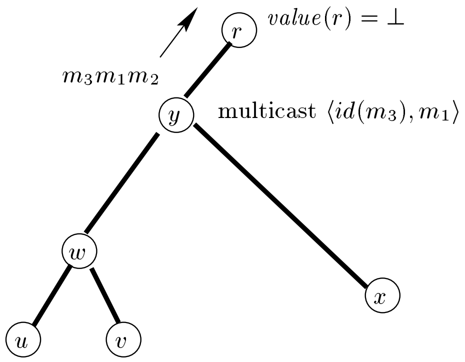

Figure 8: Combining multicast. m_1m_2 and m_3 combine. m_1 is multicast as m_3's successor. $m_3m_1m_2$ continues towards the root.

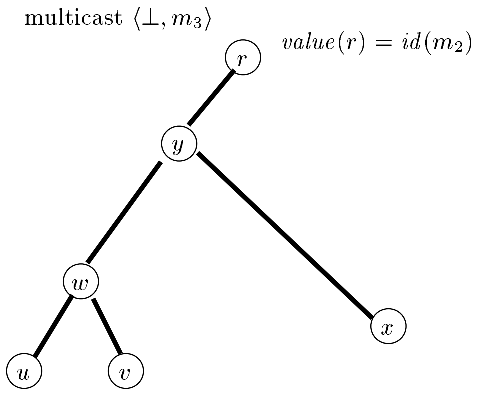

Figure 9: Combining multicast. $m_3m_1m_2$ reaches the root. m_3 is multicast as the successor of $\perp$ (i.e. m_3 is the first message in the total order). The value $id(m_2)$ is swapped into $value(r)$.

6 Discussion

In this section, we briefly compare the two ordered multicast protocols. We also discuss how to enter and leave multicast groups and fault tolerance.

6.1 Comparison

The arrow multicast has a useful *locality* property: swap-related message traffic occurs only on links between multicasting nodes, so if there is only one multicasting node, there is no swap-related traffic. In the combining tree, however, all swap-related message traffic passes through the root, even if there is only one multicaster.

6.2 Entering and Leaving

We now briefly discuss how a node might enter or leave a multicast group. Both the arrow and the combining protocols use spanning trees for connecting the member nodes. In either protocol, joining the group is straightforward: a node u just links itself as a leaf adjacent to any node v already in the tree, and informs v that it has done so. In the arrow protocol, u's arrow points to v, while in the combining protocol, v is u's parent in the tree.

Leaving a group is similar. In the arrow protocol, node u first "locks" its immediate neighbors in the tree, ensuring that message traffic between u and the neighbors has quiesced. If u is a sink, it chooses a neighbor v and makes v a sink in its place, setting $\mathit{value}(v) = \mathit{value}(u)$ and $\mathit{link}(v) = v$. For each neighbor w such that $\mathit{link}(w) = v$, it sets $\mathit{link}(w) = \mathit{link}(u)$. Node u then unlocks its neighbors and leaves the group. The combining protocol is similar, except that the sink issue does not arise.

The important point is that group membership changes are *local* operations whose complexity depends on the degree of the node, not the size of the multicast group.

6.3 Fault Tolerance

Fault tolerance is the subject of current research. It is straightforward to incorporate redundant links in either the arrow or combining tree protocols, permitting messages to circumnavigate some failed nodes. In the arrow protocol, however, it is more difficult to tolerate a sink node crash, and in the combining tree protocol, a root node crash.

We think that the challenge here is to define sensible failure semantics, specifying the kind of behavior that can occur in the presence of node and link failures. We favor the following variation of the "total order with gaps" guarantee. If m_0 and m_1 are both delivered at nodes u and v, then they are of course delivered in the same order at both. If m_0, m_1 and m_2 are delivered at u, but only m_0, and m_2 at v, then the application at v is notified that there may be one or more missing messages between m_0 and m_2. This kind of fault-tolerance is consistent in spirit with that currently provided by modern reliable multicast protocols.

6.4 Conclusions

We have found the notion of distributed swap to be helpful in designing and analyzing scalable ordered multicast protocols. We think this approach merits more attention from the research community, and we hope to convince others to explore this approach.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks. Journal of the ACM, 41(5):1020–1048, September 1994.

[3] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the International Conference on Distributed Computing Systems*, 1999.

[4] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[5] M. Demmer and M.P. Herlihy. The arrow directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing*, September 1998.

[6] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level fram-

ing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[7] Gottlieb and Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, 9, 1981.

[8] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[9] M. Herlihy and M. Warres. A tale of two directories: Implementing distributed shared objects in java. In *ACM Java Grande Conference*, June 1999.

[10] J.C.Lin and S.Paul. Rmtp: A reliable multicast transport protocol. In *Proceedings of IEEE INFOCOM*, pages 1414–1424, 1996.

[11] B. N. Levine and J.J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems Journal (ACM/Springer)*, 6(5), August 1998.

[12] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, April 1996.

[13] S.Deering. Host extensions for ip multicasting. *RFC 1112*, August 1988.

[14] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, November 1996.

[15] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in horus. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, August 1995.

[16] Roger Wattenhofer. *Distributed Counting: How to Bypass Bottlenecks.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 1998.