

# Causal Group Multicast: A Formal Description

P. Tyler

Basser Department of Computer Science  
University of Sydney  
Sydney, N.S.W. 2006, Australia

## Abstract

*This paper presents a specification of causal group multicast. Firstly causal ordering will be explained and explored in the context of group multicast. This will then be formally specified. We will then go on to discuss the effect of introducing failures into the specification. The models are expressed using an I/O automaton.*

## 1. Introduction

In this paper we present a formal specification for causal group multicast systems assuming no failures. We will then briefly discuss what effect crashes may have on such a system. This has arisen from work in investigating the ISIS[1][2] multicast protocols.

In the past specifications may have been in a language open to some interpretation, but here we provide a precise specification using an I/O Automaton[6]. In Fekete[4], it is argued that formal methods can play an important part in specification and exploration of a service. Further it is suggested that a user of a service relies on "what is provided" rather than "how it is provided". The specification gives the user a precise definition of what service is provided and can be used as a tool for a system designer in verifying compatibility of newer algorithms.

Multicast is a network service similar to broadcast, except that it restricts message delivery to a specific set of nodes. The delivery set can be defined in many ways, including requiring the sending client to provide a list of nodes at which to deliver the message, or by the service providing a grouping mechanism. A grouping mechanism is simply a single identification for a group of nodes. The nodes in the group will usually have a common task or purpose and communicate using a unique group identification (group id). A node is free to join and leave the group, with the underlying communication protocol keeping track of which nodes are currently members of the group. To send a message, a sending node need only supply the message and the group identification. ISIS

requires that a node belong to a group to which it sends a message, and this is what we model. Using groups assumes that the node knows the group id or is able to get it independently of the multicast algorithm. Usually we expect the group id would be agreed to by the client designers.

Let us consider an example where such a service is of benefit. Consider a replicated database service such as the archie service available on the internet for finding files available by anonymous ftp. Due to high loads a single site has not been sufficient and replication has taken place. (Although archie doesn't use a group multicast system for propagation of its database, we shall use it as an example of where a group multicast would be of benefit.) Nodes interested in maintaining the database simply join the group for database updates, and start receiving database information. (We are assuming that an initial version is requested at some previous stage.) Sending nodes need not know of the new replica explicitly as the underlying package provides the necessary service through the use of groups. The new replica is now able to serve other nodes that have database search requests. Moreover, the new node can be given the responsibility to collect information about sites close to it and to submit that information to the system by a multicast to the group.

Although multicast in itself is useful, clients often require message ordering properties. The ISIS system offers two ordering properties, atomic and causal. Atomic ordering places a strict order on all messages, making it possible to determine a global sequence of all messages and this is the order used for delivery. By using this facility, clients know that the order in which they receive messages from the atomic service is the same order as others in the group. Causal ordering however is less strict than atomic and is based on the "happens-before" relation of Lamport[5]. The idea of causal multicast is that if knowledge from the contents of a message, say  $m_1$  could have affected another, say  $m_2$ , then  $m_2$  causally depends on  $m_1$ . We then require  $m_1$  to be delivered before  $m_2$ . Figure 1(a) shows how delivery of  $m_2$  may have to be delayed. Should two nodes send messages where neither message causally

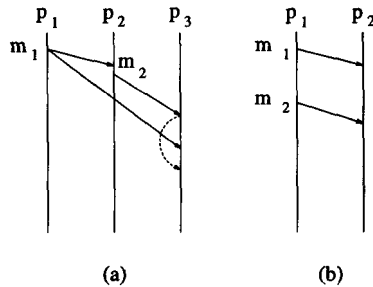


Figure 1. Direct Causal Dependence

depends on the other, then a causal multicast will not place any particular order on the two messages. More formally then, if client  $p_2$  receives  $m_1$  from the service and then sends  $m_2$ ,  $m_2$  causally depends on  $m_1$ . This is found in figure 1(a) at node  $p_2$  and demonstrates a delay of  $m_2$  at  $p_3$ . Also if  $p_1$  sends  $m_1$ , then later sends  $m_2$  as in figure 1(b),  $m_2$  causally depends on  $m_1$ . Either of these situations are referred to as being a "direct dependence". Causality can extend for many messages, thus causal dependency is taken to be the transitive closure of the direct dependency.

The two concepts of causal ordering and group multicast can be combined, ISIS being an example of this. In building a causal group multicast system, designers are confronted with the decision of whether to provide causal ordering between groups as well as within groups. Often this decision is based on the cost of providing full causal orders, the usual cost being larger message headers. ISIS however does provide causal ordering between groups despite the higher cost and so we have chosen to model this.

The rest of this paper presents a specification of causal group multicast. In Section 2 we introduce the I/O automaton and then give a specification of causal group multicast. Section 3 briefly discusses failures and their affect on the specification. The final section gives a summary and describes further and related work being undertaken.

## 2. Causal multicast with groups: no failures

### 2.1 I/O automata

Our specification will be presented in the form of an I/O automaton[6]. There are three types of actions, input actions generated by the environment, output actions generated by the automaton that transmit to the environment, and internal actions which are within the I/O automaton itself. Internal and output actions have preconditions and effects while input actions only have effects. A precondition must be true before an effect

can occur, but as input actions have no precondition they can occur anywhere in an execution. It is important to note that just because the precondition is true doesn't mean the action has to occur immediately. When an action does occur though, the effect of the action changes the state of the automaton. Such a change occurs in a discrete step, that is the whole effect transforms the state of the automaton in one step. This property is known as atomicity and actions are said to be atomic. It is important to stress that although the effects presented later appear like code with a sequence of assignments, the whole code for the effect occurs in a single step. An execution of an I/O automaton is an alternating sequence of states and actions where the action's precondition is true in the previous state of the execution. For a complete description of I/O Automata see Lynch[6].

### 2.2 Specification of group multicast without crashes

In this section we give the I/O Automaton CM, a specification of the task we expect a group causal multicast system to perform. In CM we assume that there are no failures in either the nodes or the network. With this assumption we can insist that a message be delivered at all nodes in the group. However as the group membership fluctuates, our concept of which specific nodes receive the message is a little unclear. The purpose of the specification is to clarify what users expect of such a service in this situation. In the specification, sets of nodes for a group are views, and successive views for a group will change by the addition or deletion of one node only.

The service described by CM involves communication between a client and the service. We have been using the term node to describe the client but this could equally be a process or other entity. The client needs to be able to send and receive messages, as well as join and leave groups. This is achieved through input and output actions. For sending a message, the client has to supply a message and the group to which the message is being sent. The action for sending a message  $m$  in group  $g$  is  $CMcast\_send(m, g, p)$ , and is shared between the client at  $p$  and the service. For the automaton CM, this action is an input action which conveys message  $m$  and the group id  $g$  from the client to the service. (If we were to model the client explicitly this action would be an output action of the client model.) Similarly for message delivery there is the output action of CM (and an input action of the client)  $CMcast\_rcv(m, p, q)$ . This action delivers the message  $m$  sent by node  $p$  to node  $q$ .

Group changes are achieved with the input actions  $CMadd\_client(p, g)$  and  $CMremove\_client(p, g)$ . These actions are requests for client  $p$  to be added or

removed from group  $g$ . The effect of these actions is not immediate on the group. Although the specification could provide an immediate change to the group, this property cannot be provided by a distributed algorithm. For example, if two clients,  $p$  and  $q$ , cause the input actions  $CMadd\_client(p,g)$  and  $CMadd\_client(q,g)$  respectively, two new views will be created. An immediate change would make it possible for node  $p$  to be operating in an intermediate view without node  $q$ , and node  $q$  to operate without node  $p$ . It is the responsibility of the algorithm to prevent this by either placing some global ordering on view changes or disabling the intermediate views. Either way a delay will be introduced. Our final action of CM is an internal action named  $CMnew\_view(p)$ . As the name suggests this action is responsible for causing the next view change in client  $p$ 's list of requests.

The automaton CM keeps significant information in global variables. In our model arrays are often indexed by non-negative integers and so are infinite, allowing us to assign a unique index to each message. This index is used to reference elements in the arrays holding information about the message. A couple of arrays are indexed by other means and this is noted when they are explained.

When a message enters CM it is assigned an index (say  $i$ ). The message is stored in  $msg[i]$ , the source node in  $src[i]$ , and the group identification in  $grp[i]$ . The array  $dest[i]$  contains an index for the view array (mentioned below) indicating the view of  $grp[i]$  in which the message is sent.  $depend[i]$  is a list of message indexes, being the messages on which message  $i$  causally depends. We use  $deliv[i]$ , a set of client id's, to keep track of where the message has been delivered. The initial values of these arrays are not important as the elements are set on a  $CMcast\_send(m,p,g)$  action. The variable  $next$  is used to indicate the next index for storing a multicast and is initially 0.

The remaining variables are used for group changes. The  $view$  array is indexed by a non-negative integer and a group identifier. We notate this as  $view[i,g]$ , representing the set of nodes in the  $i$ th view of group  $g$ . These sets are initially empty. We stated earlier that the input actions to change group membership do not immediately cause a change to the view but that the change is delayed. In the model this is achieved by keeping lists of group change requests for each node in the array  $view\_change$ , where  $view\_change[p,i]$  is the  $i$ th change request for node  $p$ . These are triples of the form  $(ACTION,node,group)$  where  $ACTION$  is either  $ADD$  or  $REMOVE$ . It is initially empty for each node. The  $view\_change$  arrays act as a list. The variable  $view\_in[p]$  holds the index for the position of the next element to enter the list for

node  $p$ , while  $view\_out[p]$  is the index for the next outgoing element and both are initially 0. If  $view\_in[p]$  equals  $view\_out[p]$  then the list for node  $p$  can be considered empty. Our final variable is  $next\_view[g]$ , an array indexed by a group id. This array holds the index for the next view for group  $g$  and is initially 0, for all groups.

All these variables are global, with all information available to all the nodes. As we are providing a specification and not modelling a specific implementation or algorithm, we do not have to restrict ourselves to using local variables. The use of localised variables would introduce complexities that are found in real algorithms. We avoided this in specifications to keep them as simple and concise as possible. It is important to remember that the specification is to state what is done, not how it is achieved.

In notating the transition relation, the set of actions with preconditions and effects, we notate the state before the action as  $S'$ , while  $S$  is the state afterwards. (This is the usual convention in I/O automata and is the reverse to that used in Z) To notate  $src[i]$  in state  $S'$ , we use  $S'.src[i]$ . The transition relations stated below use a function for determining the set  $depend[i]$ . Define

$$direct(i,p,S) = \{j : j < i \wedge (p \in S.deliv[j] \vee p = S.src[j])\}.$$

(With appropriate  $S$ ,  $direct(i,p,S)$  will be the set of message indexes on which message  $i$  has a "direct dependence".) Also define

$$depend(i,p,S) = direct(i,p,S) \cup \left( \bigcup_{j \in direct(i,p,S)} S.depend[j] \right).$$

If for all new messages we use  $depend(i,p,S)$  with  $S$  being the state before a  $CMcast\_send$  (where  $i$  is the index of the new message and  $p$  the sending node id) then the set  $depend(i,p,S)$  is the set of message indexes on which the new message causally depends.

The transition relation for CM is determined by the code in figure 2. The code only notates the changes from  $S'$  to  $S$ , and all other variables remain unchanged. Each action is presented with a brief explanation.

### 3. Causal group multicast with failures

In this section we describe how to change the model to include failures. Previously we guaranteed delivery of every message. Unfortunately with failures this cannot be guaranteed, and a specification would have to model the possibility of message loss either by a client's failure or by network failure. It would be desirable if the system behaved in the same way to all remaining clients, that is that a causal group multicast

**Action:**  $CMcast\_send(m, p, g)$

Description: Store incoming message  $m$  and group  $g$ , choose a view and set causal dependencies.

Effect:

$S.msg[S'.next] = m$  { Add message, source node }  
 $S.src[S'.next] = p$  { and destinations to appropriate arrays }  
 $S.grp[S'.next] = g$   
 $S.dest[S'.next] = v$  where  $v$  is any value satisfying: { Choose a view }  
 $- 0 \leq v < S'.next\_view[g]$  { View must be valid }  
 $- p \in S'.view[v, g]$  { Source node must be in the view }  
 $- \forall k: ((k \in depend(S'.next, p, S')) \wedge (S'.grp[k] = g)) \Rightarrow S'.dest[k] \leq v$  { View meets causality }  
 $S.depend[S'.next] = depend(S'.next, p, S')$  { Set causal dependencies }  
 $S.deliv[S'.next] = \emptyset$  { Not delivered anywhere }  
 $S.next = S'.next + 1$  { Increase index }

**Action:**  $CMcast\_rcv(m, p, q)$

Description: The precondition checks the conditions for delivery, including causality.

Precondition:

$0 \leq i < S'.next$  { Message is in valid part of array }  
 $m = S'.msg[i]$  { Correct corresponding message }  
 $q = S'.src[i]$  { Correct source }  
 $p \in S'.view[S'.dest[i], S'.grp[i]]$  {  $q$  is a destination }  
 $p \notin S'.deliv[i]$  { Delivery at  $q$  has not yet occurred }  
 $\forall j: ((j \in S'.depends[i]) \wedge (p \in S'.view[S'.dest[j], S'.grp[j]])) \Rightarrow (p \in S'.deliv[j])$   
 { Check causality requirements }

Effect:

$S.deliv[i] = S'.deliv[i] \cup \{q\}$  { Mark delivery }

**Action:**  $CMadd\_client(p, g)$

Description: Add client  $p$  to the group  $g$

Effect:

$S.view\_change[p, S.view\_in[p]] = (ADD, p, g)$  { Request client  $p$  added to group  $g$  }  
 $S.view\_in[p] = S'.view\_in[p] + 1$

**Action:**  $CMremove\_client(p, g)$

Description: Remove client  $p$  to the group  $g$

Effect:

$S.view\_change[p, S.view\_in[p]] = (REMOVE, p, g)$  { Request client  $p$  be removed from group  $g$  }  
 $S.view\_in[p] = S'.view\_in[p] + 1$

**Action:**  $CMnew\_view(p)$

Description: Install group change requested previously.

Precondition:

$S'.view\_out[p] < S'.view\_in[p]$   
 $(a, p, g) = S'.view\_change[p, S'.view\_out[p]]$  { Next change for  $p$  }

Effect:

if  $a = ADD$  then { Add client to group  $g$  }  
 $S.view[S'.next\_view[g], g] = S'.view[S'.next\_view[g], g] \cup \{p\}$   
 else if  $a = REMOVE$  then { Remove client from group  $g$  }  
 $S.view[S'.next\_view[g], g] = S'.view[S'.next\_view[g], g] - \{p\}$   
 endif  
 $S.view\_out[p] = S'.view\_out[p] + 1$   
 $S.next\_view[g] = S'.next\_view[g] + 1$  { Change next view index }

Figure 2. Transition Relation for CM

will either deliver a message to all surviving group members, or not to any surviving members. This property gives us uniform behaviour for all operational nodes in the group and is what ISIS guarantees.

Firstly, let us consider an example where clearly a message won't be delivered. Two clients,  $p_1$  and  $p_2$ , belong to group  $g$ . The service receives a message,  $m_1$  from  $p_1$  to be sent to group  $g$ , but  $p_1$  crashes before there is time to transmit. Clearly then,  $m_1$  will not be delivered anywhere and it is as though  $m_1$  was never sent. Now consider the situation found in figure 3(a). Note that the figure shows delivery to the causal group multicast algorithm and not delivery to the client. The group consists of 3 nodes,  $p_1$ ,  $p_2$ , and  $p_3$ . Message  $m_1$  had been transmitted by  $p_1$  to  $p_2$  but then  $p_1$  crashes and is unable to send it to  $p_3$ . Client  $p_2$  receives and delivers  $m_1$  so we expect  $m_1$  to be sent by  $p_2$  to  $p_3$ . Should  $p_2$  crash without having forwarded  $m_1$  on, then clearly  $m_1$  has been lost. Other messages could also be lost because a message on which it is causally dependent has been lost. For example in 3(b), as before  $p_2$  delivers  $m_1$  but then sends  $m_2$  to  $p_3$ . Node  $p_2$  then crashes. We have  $m_2$  awaiting delivery at  $p_3$  but it is dependent on  $m_1$  which has been lost. We could also consider  $m_2$  as lost.

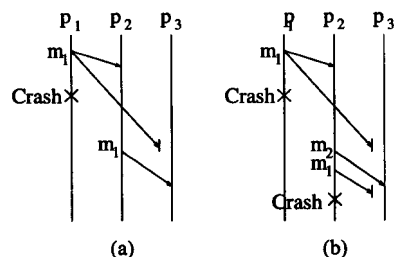


Figure 3. Crash examples

Another issue that arises out of failures is failure detection. Detecting failure within a network system is an imprecise task. Using techniques such as timeouts makes it impossible to detect between a machine failure and network failure. A timeout could be the symptom of a heavy load on a processor or a heavily loaded network link, rather than a failure. So in a model it is not enough just to model a crash. We also have to allow for a form of crash detection mechanism to notify of perceived crashes. Note that just because a node has been perceived to have failed isn't an indication of actual failure. The issue of crash detection is discussed in great

detail by Ricciardi[7]. The detected failures need not be real failures and in fact can be a symptom of a network crash so we choose that a false failure detection will force a client to leave its groups. A full presentation and discussion of a specification reflecting this can be found in Tyler[8].

#### 4. Conclusions and Further Work

This paper has presented a specification for causal group multicast without failures. We then outlined the issues for extending the model to include failures. It was suggested that such a specification would be useful in providing a clear picture of the service, helping in determining compatibility and useful for producing formal proofs.

Further work will include modeling the ISIS system using I/O automata. A proof will then be constructed showing that ISIS provides the service of an extended specification which includes failures.

#### References

- [1] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures", *ACM TOCS* Vol. 5, No. 1, February 1987, pp 47-76.
- [2] K. Birman, A. Schiper and P. Stephenson, "Lightweight Causal and Atomic Group Multicast", *ACM TOCS* Vol. 9, No. 3, August 1991, pp 272-314.
- [3] J. Chang and N. Maxemchuk, "Reliable Broadcast Protocols", *ACM TOCS* Vol. 2, No. 3, August 1984, pp 251-273.
- [4] A. Fekete, "Formal Models of Communication Services: A Case Study", *IEEE Computer*, August 1993, pp 37-47.
- [5] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. ACM*, Vol. 21, No. 7, July 1978, pp 558-565.
- [6] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata", *CWI Quarterly*, Vol. 2, 1989 pp 219-246.
- [7] A. Ricciardi and K. Birman, *Using Process Groups to Implement Failure Detection in Asynchronous Environments*, Cornell University Computer Science Department Technical Report TR91-1188.
- [8] P. Tyler *Describing Causal Group Multicast with Failures*, University of Sydney, Basser Department of Computer Science Technical Report, To appear 1994.