# The University of Sydney

# Casual Group Multicast: A Formal Description

Technical Report Number 484

July 1994

Paul Tyler

ISBN 0 86758 927 2

Basser Department of Computer Science
University of Sydney NSW 2006

# 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 two specifications of causal group multicast. In the first, causal ordering will be explained and explored in the context of group multicast. Failures will be introduced and a specification of causal group multicast which includes node crashes will be given. This final specification describes the service provided by ISIS's causal multicast facility. The models are expressed using the I/O automaton.*

## 1. Introduction

This paper presents formal specifications for causal group multicast systems, one assuming no failures, the second allowing the possibility of real node failures and perceived node/link failures. The second specification is a specification for the ISIS[1][2] causal multicast service.

In the past specifications may have been given in languages open to some interpretation, but here we provide a precise specification using the 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 a service provides, 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, knowing that the nodes who require the message have joined the group. A node is responsible for joining or leaving a group and is not restricted to belonging to one group. ISIS however 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. Clearly a single site has not been sufficient as too many people wish to use the service and as such the service became overloaded. Replicating the database at other sites reduced the load. (Although archie doesn't appear to 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. As the new replica is now receiving up to date information, it can serve other nodes that have database search requests. Moreover, the new node upon joining could be given the responsibility of collecting new information about sites close to it and to submit that information to the system by a multicast to the group. Such a system would provide the replicated database at a number of sites, spreading the load across multiple nodes.

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. This facility can be important to many applications, but it can be expensive to provide and not all clients have such a strict order requirement. 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 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 situation 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". In addition, information about $m_1$ could be contained in $m_2$ which causes $m_3$. Such a line of 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. Despite the cost however ISIS does provide causal ordering between groups and so we have chosen to model this.

The rest of this paper presents specifications of causal group multicast. In Section 2 we introduce the I/O automaton and then give a specification of causal group multicast.

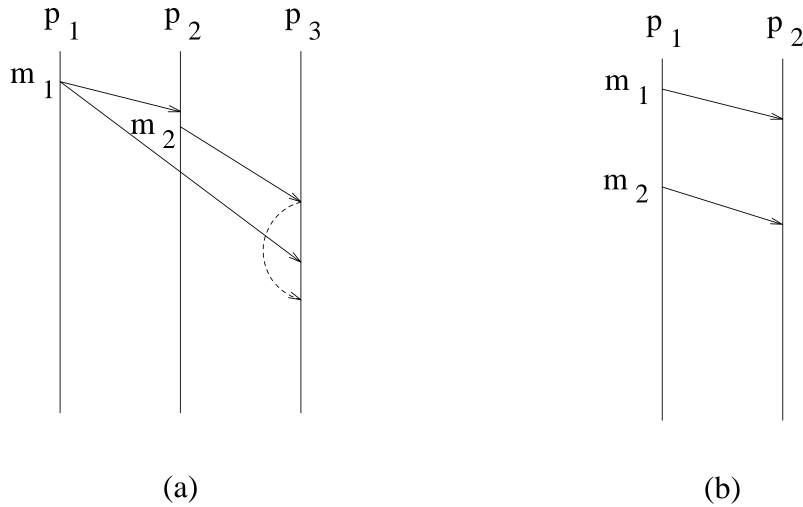


**Figure 1.** Direct Causal Dependence

Section 3 introduces possible failures to the system, formally specifying when it is possible a message could be lost. The final section gives a summary and describes further 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 the I/O Automaton 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 users can expect a group causal multicast system to perform. In CM we assume that there are no failures in either the nodes or the network. This means that a node cannot crash and performs normally throughout the entire execution. With this assumption we can insist

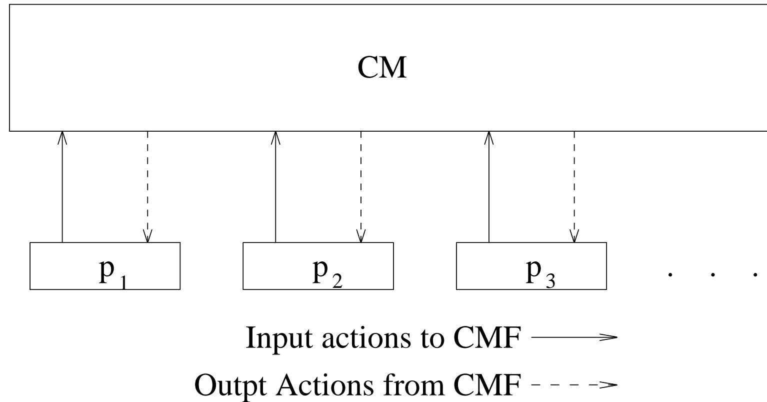


**Figure 2.** Client - Service Relationship

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 we 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. These lines of communication are illustrated in figure 2. 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, 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. If the change were immediate though, it would be 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 created by installing or removing *p* based on its previous 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 significant as they 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 ith 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 ith change request for node p. These are triples of the form $(ACTION,p,g)$ where *ACTION* is either *ADD* or *REMOVE*, and is a request to either add or remove the client $p$ to or from a group $g$ respectively. 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 the 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

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

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

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

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 3. The code only notates the changes from $S'$ to $S$, and all other variables remain unchanged. Each action is presented with a brief description.

**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 */

- $0 \le v < S'.next_view[g]$ /* View must be valid */
- $p \in S'.view[v,g]$ /* Source node must be in the view */
- $\forall k:((k \in depend(S'.next,p,S')) \wedge (S'.grp[k]=g)) \Rightarrow S'.dest[k] \le v$ /* 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:

$0 \le i < S'.next$ /* 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:

$S.deliv[i]=S'.deliv[i] \cup \{p\}$ /* Mark delivery */

**Action:** $CMadd_client(p,g)$

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

Effect:

$S.view_change[p,S.view_in[p]]=\{(ADD,p,g)\}$

$S.view_in[p]=S'.view_in[p]+1$

**Action:** $CMremove_client(p,g)$

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

Effect:

$S.view_change[p, S.view_in[p]] = \{(REMOVE, p, g)\}$

$S.view_in[p] = S'.view_in[p] + 1$

**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 3.** Transition Relation for CM

### 3. Causal Group Multicast with Failures

In this section we introduce failures to causal group multicast. Our previous specification guaranteed delivery of every message. Unfortunately with failures this cannot be guaranteed. So in specifying causal group multicast with failures, we have to model the possibility of message loss caused by client or by network failure. It would be desirable if the system behaved in the same way to all remaining operational clients, that is that a causal group multicast service 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 as in figure 4(a). (Note that the figure shows sending from and delivery to the causal group multicast algorithm and not the client.) Three clients, $p_1$, $p_2$ and $p_3$ belong to group $g$. The service receives a message, $m_1$ at $p_1$ to be sent to group $g$. Unfortunately before completing transmission to either node, $p_1$ crashes. 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 4(b). The group consists of 3 nodes, $p_1$, $p_2$, and $p_3$. Message $m_1$ has been given to the service at $p_1$ for causal multicast to the group. Suppose $p_2$ and $p_3$ are unable to be reached by the underlying communication protocol simultaneously leading to the use of point to point communication. Node $p_1$ transmits to $p_2$ but crashes before having the opportunity to send to $p_3$. Client $p_2$ receives and delivers $m_1$ so to keep consistency we expect $m_1$ to be forwarded by $p_2$ to $p_3$. Should $p_2$ crash without having

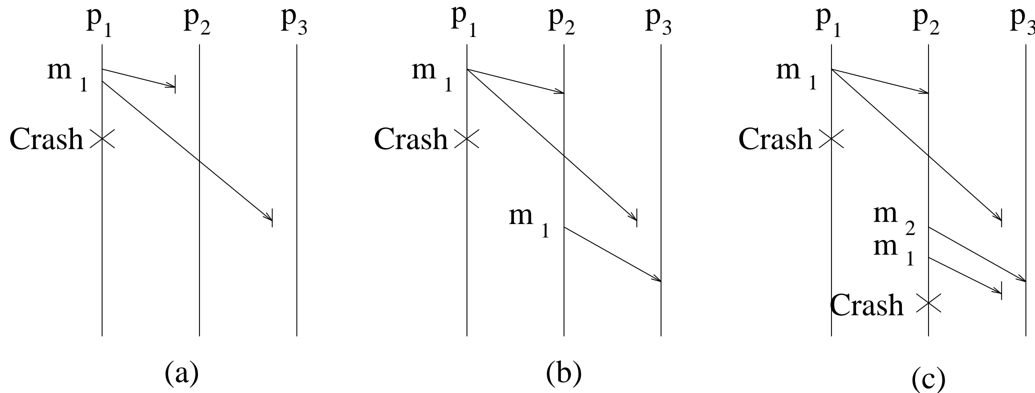


**Figure 4.** Message Loss

forwarded $m_1$ on, then $m_1$ has been lost. Other messages could also be lost because a message on which it is causally dependent has been lost. For example in 4(c), 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 will never be delivered at $p_3$. It is the choice of ISIS and so of our specification to consider $m_2$ lost in this case.

Another issue that arises out of failures is failure detection. Detecting failure within a network system is an imprecise task. Timeouts, for example, may be a symptom of machine or network failure, but the timeout could also be the symptom of a heavy load on a processor or a heavily loaded network link. So in our model it is not enough just to model a crash. We also have to allow for a form of crash detection mechanism to notify the service of perceived crashes. In this situation just because a node has been perceived to have failed isn't an indication of actual failure. We do not model a specific crash detection mechanism, but define an input action of the specification that notifies the service of such a perceived crash. This input action would then be caused by the crash detection algorithm. As network failures will look like node failures, our failure detection of the network will be by notification of perceived failures of unreachable nodes. The issue of crash detection is discussed in great detail in Ricciardi[7].

### 3.1 Specification of Group Multicast with crashes

Here we give the I/O Automaton CMF, a specification for group causal multicast with client failures and detected failures. The detected failures need not be real failures and in fact can be a symptom of a network crash, though a false failure detection will force a client to leave it's groups. We make two assumptions about failures. Firstly that once a client has failed or has been perceived as having failed, it is considered as having crashed for the rest of the execution and will not rejoin using the same identification. (It could however rejoin as though it were a new client by using a new unique identification.) Our second assumption is that a node fails gracefully and that there are no malicious actions taken by the node after a crash.. We also make two assumptions of the detection system. If a node does crash, the failure will be detected. Secondly if one node receives a crash detection notification of node $p$, whether a real crash or not, then all other nodes in groups with $p$ will also receive this notification.

Our model CMF is similar to CM. Once again groups pass through a series of views which change by the addition or deletion of one node with the extra property in CMF that a deletion may be caused by another node detecting a crash. Our input actions are $CMFcast_send(m,p,g)$, $CMFadd_client(p,g)$ and $CMFremove_client(p,g)$, and they serve the same purpose as the corresponding actions in CM. There are two additional input actions, $CMFcrash(p)$ and $CMFcrash_detect(p,q)$. $CMFcrash(p)$ is an action representing a real crash at client *p*. It is assumed that this action will be the last from client *p* and basically stops any further internal action by *p*. It is as though *p* suffered a power failure at that point in the execution. The other new input action, $CMFcrash_detect(p,q)$ is from client *p* notifying the detection of a crash at client *q*. It is assumed that the crash detection system will eventually cause a $CMFcrash_detect(p,q)$ after a $CMFcrash(q)$ at each *q* which shares a group with *p*.

An internal action, $erase_msg(i)$, has been added. It is responsible for causing the deletion of messages in the system when the appropriate nodes have crashed. A new action $CMFnew_crash_view(p)$ has also been added, allowing for a view change caused by a crash. It behaves in a similar manner to $CMFnew_view$.

The variables for CMF are as in CM with the addition of the following variables. To notate a client's state as being either operational or failed, we use $state[p]$, which is set to *CRASHED* when the node crashes. In the starting state $state[p]$ is set to *OK* for all nodes. Our other new variable uses the same index as the message (say i). $erased[i]$ will be set to *YES* when $msg[i]$ gets deleted from the system due to a crash. The initial value of $erase[i]$ is *NO*. Each node keeps a set of nodes that have been detected as crashed in the set $crashed[p]$, which is initially empty.

The transition relation for CMF is determined by the code in figure 5. We use the function *depend* as it was defined earlier for CM. Many of the actions of the transition relation for CMF are similar to that of CM.

**Action:** $CMFcast_send(m,p,g)$

Description: Store the incoming message *m* in group *g*, choose a view and set causal dependencies

Effect:

if $S'.state[p] \neq CRASHED$ then

$S.msg[S'.next] = m$ /* Add message, source node and */

$S.src[S'.next] = p$ /* destinations to appropriate arrays */

$S.grp[S'.next] = g$

$S.dest[i] = v$ 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 of message meets causality */

$S.depend[S'.next] = depend(S'.next,p,S')$ /* Set causal dependencies */

$S.deliv[S'.next] = \emptyset$ /* Not delivered anywhere yet */

$S.next = S'.next + 1$ /* Increase index */

fi

**Action:** $CMFcast_rcv(m,p,q)$

Description: Deliver message *m* from *q* to *p*. The precondition guarantees causality.

Precondition:

$0 \le i < S'.next$ /* 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 */

$S'.state[p] \neq CRASHED$

$S'.erased[i] = NO$

Effect:

$S.deliv[i] = S'.deliv[i] \cup \{p\}$ /* Mark delivery */

**Action:** $CMFadd_client(p,g)$

Description: Schedule the addition of client *p* to group *g*.

Effect:

$S.view_change[p, S.view_in[p]] = \{(ADD, p, g)\}$

/* Request client p added to g */

$S.view_in[p] = S'.view_in[p] + 1$

**Action:** $CMFremove_client(p,g)$

Description: Schedule the removal of client *p* to group *g*.

Effect:

$S.view_change[p, S.view_in[p]] = \{(REMOVE, p, g)\}$

/* Request client p be removed from g */

$S.view_in[p] = S'.view_in[p] + 1$

**Action:** $CMFcrash_detect(p,q)$

Description: Record the detection by client *p* of the crash of client *q*.

Effect:

$S.crashed[p] = S'.crashed[p] \cup \{q\}$ /* Add q to p's list of crashed nodes */

if $p = q$ then /* If this is a self detection */

$S.state[p] = CRASHED$ /* mark p as crashed */

**Action:** $CMFcrash(p)$

Description: Record the physical crash of client *p*.

Effect:

$S.state[p] = CRASHED$ /* Mark p as having crashed */

**Action:** $CMFnew_crash_view(p,q)$

Description: Create a new view without client $q$. This is the result of a previous $CMFcrash_detect$.

Precondition:

$q \in S'.crashed[p]$ /* p has perceived q has failed */

$\exists g: \{p,q\} \subseteq S'.view[S'.next_view[g]-1, g]$ /* p and q in recent view */

Effect:

$S.view[S'.next_view[g]] = S'.view[S'.next_view[g]] - \{q\}$

/* Remove q from group */

$S.view_change[q] = \varnothing$ /* Don't allow any more changes */

$S.state[q] = CRASHED$ /* Mark q as crashed */

**Action:** $CMFnew_view(p)$

Description: Install a previously requested group change.

Precondition:

$S'.view_out[p] < S'.view_in[p]$

$(a,p,g) = S'.view_change[p, S'.view_out[p]]$ /* Next change from client 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 */

**Action:** $CMFerase_msg(i)$

Description: Cause the erasure of the message with index $i$.

Precondition:

$\forall q: q \in S'.deliv[i] \Rightarrow S'.state[q] = CRASHED$

/* Only delivered at crashed nodes */

$S'.state[S'.src[i]] = CRASHED$ /* Source has crashed */

$S'.erase[i] = NO$ /* Message not yet erased */

Effect:

$S.erased[i] = YES$ /* Erase message */

**Figure 5.** CMF Code Segment

The final action $CMFerase_msg(i)$ is the fundamental action for erasing a message after a failure. It requires that the source node and all nodes that have delivered the message have crashed. As stated earlier, just because the action's precondition is true doesn't mean the action occurs immediately. Fairness of an I/O automaton means that provided the precondition remain true from a state onwards in an execution, the action will eventually occur. In this model it means that either a $CMFerase_msg(i)$ or a $CMFcast_rcv(m,p,g)$ for $msg[i]=m$ will occur first, and this will cause the

precondition of the other action to become false. This is how the non-deterministic nature of message loss is expressed.

#### 4. Conclusions and Further Work

This paper has presented two specifications for causal group multicast, one being for an algorithm for causal broadcast without failures, a second with failures describing the service provided by causal group multicast as found in ISIS. 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 specified in CMF.

#### 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", To appear in IEEE Computer
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.