Distributed Collaboration

Manikya Singh, Rajat Garg, Vishal Tomar

Indian Institute of Technology, Kharagpur, India

Introduction

Any system that supports the interaction of two or more spatially located (not necessarily) computers is considered as distributed collaborative system. The term collaborative refers to the group of computers working together and the distributed refers to their sharing of resources instead of sharing memory. With the advancement in IT sectors and requirements for distantly located people to work together, this systems are getting revised to improved and updated to deal with different upcoming user-specific demands. These demands can vary from instant messaging to group conferencing systems.

In distributed collaborative systems, operations or data strems from each user is multicasted to other peers in the group, based on the information received by a node from it's peers it compute what updates are required to be done on the UI and what data stream to output.

Design Issues

Multicast

To make system independent of underlying network, Multicast activity must be implemented at Application layer instead of Network layer. If all data from each node is sent directly to other nodes, it will result in congestion in network due to outburst of data packets. Each node will have to deal with (n-1) messages if their are n nodes in the group which results in low scalability. A centralized approach is also not very scalable.

Peers self organize themselves into logical topology called overlay network over which data is transmitted. In a tree based organization, number of message each node has to deal with messages is equal to its degree. But a method for dynamic updation of this topology is required as new peers can join or leave any time, this updation protocol also include membership protocols for the group.

NICE protocol[7] is a scalable application layer protocol in which peers are arranged in 4-clique clusters, each cluster has a leader, these leaders again form 4-clique clusters, only one leader is present at highest layer in the network.

Ordering of Events

All nodes must agree on the order in which events occurs in the system, when a node receives a message from other nodes, it needs to know in which order did the events took place, or in which order will other nodes process those two events. This problem is solved by use of Lamports logical clock, which can be used to design protocols for casual, atomic, atomic cousal delivery of messages.

In atomic causal delivery order, all events in the system are executed in total order, but it using such protocol will lead to slow response time of the system, since commands executed by the system at a node will not be executed as they are made. Hence, we need to find a way to deliver messages to an application in causal order but still preserve total ordering after all the transit messages have been delivered to recipient nodes.

Concurrency Control

Concurrency control problem is central to groupware systems. Each participant maintains it's copy of shared context and updates to this context when done by a participant are reflected on local application interface and notifications are sent to remaining participants. It can either be done by enforcing total ordering or equivalent partial orderings(i.e. Partial ordering which result in same final state as total ordering). However, choosing any total ordering(for example based on id of nodes) may not be well suited for application needs. Their may also be a need to drop some events execution in case of ambiguity in users actions, for example if two users concurrently deletes same character then is is most certainly the case that execution both the operations is unintended and only one delete should be performed by the system.

One basic idea is to deliver messages in causal order and undo delivery of messages which are found to be out of order and then redo them in order. A lot of undoing and redoing will result in a slow application, in order to reduce number of undos and redos, the fact that commutative events can be carried out in any order and masked events can be ignored is used. This approach can be improved further by use of operation transforms which is modifying older event not yet delivered so that it can be carried out after a younger event without need to undo. In collaborative applications like shared text buffer, two users may command to delete same part of the buffer, carrying out the operations on any order will result in deleting extra characters which was not the intent of the user.

Audio Packet Loss

It is very crucial for a collaborative audio application to achieve a very high level of audio quality, because human hearing is highly sensitive to audio distortions. Audio packet transmitting system faces audio degradation due to either the loss of packets or the late arrivals. The audio device becomes temporally idle due to packet losses or late arrivals which causes irritating clicking noises. The system is aimed to be designed in such a way that these dropouts get minimized.
To overcome audio degradation due to packet loss, the audio tool sends the audio sample as a two consecutive packets but in different audio encodings. Audio packet n contains audio sample n in the primary speech coding along with sample n - 1 in a secondary coding method. The primary coding results in a high data rate and high quality audio while the secondary coding produces lower quality audio with a much reduced data rate. A primary audio sample n can be "repaired" if the secondary sample in packet n + 1 is received.

Hands-free Mode

The voice detection in send-audio subsystem results in a "hands-free" mode of usage of the collaborative system. The voice data are converted to user-specified format(e.g GSM) and multicast to all participants in the session. A sample with amplitude less than specified voice threshold *v* is considered silent, otherwise treated as a voice. In both cases the sample is put in the circular buffer. If the current sample contains voice but not the previous one then *k1* buffered untransmitted packets are sent prior to the current one. After last detected voice packet *k2* samples are sent. The parameters *v*, *k1* and *k2* are very specific to the individual's articulation and hence set empirically. Under the circumstances, where hardware does not support full-duplex or ground noise level is very high, the system will operate in half-duplex "Press to Talk" mode with no more "hands-free" mode available.

Document Collaboration

Groupware systems are computer based systems that provide common work space through application interface to multiple users concurrently. Each member updates it's shared context via application interface In real time groupware systems, notification time, i.e. time taken to notify group members of updates, must be comparable to system's response time i.e., time taken to show updates locally. In the next two section we will look at a brief overview of design issues and on correctness of the system.

Correctness of the System

The groupware system is said to be in quiescent state when all generated operations have been performed at all the sites. All the objects in shared context must be in the same state at quiescent. This condition is called convergence property.

We can define partial ordering of all operations generated using Lamport's Vector Clocks[1], if an operation "happened before" another then it must be executed before the latter one by all the nodes in the system. Concurrent events must be executed in same(but any) order at all the nodes. This condition is called precedence property. A collaborative system should satisfy both the properties.

Application Model

As described in [2], We will model application in terms of objects, operations, events and state variables of the node.

Each node has following state variables:
me: Unique id of the node
S: Set of ids of each node
t: Logical clock of the node
seq: Sequence number of last event sent
L: Log of events delivered, stack data structure
Q: Queue of events waiting for execution, the events are sorted by logical timestamp from oldest to most recent, for concurrent events, event with lower sender id is considered older (or of higher priority)
O: Set of all objects (shared context)
obj: Function mapping object id to object, i.e, obj: ID $\rightarrow$ O $\cup$ {null, deleted}
    obj(a) = null , if no object in the context has id=a
    obj(a) = deleted, if object with id=a has been deleted from the context
    obj(a) = (a, state of a) , otherwise
lastSeq: Dictionary holding sequence number of last event received from each node.
lastClock: Dictionary holding time stamp of last event received from each node.

An object is defined as 3-tuple o=(id, st, F) , where id is unique identifier of an object and st is it's state. Each node when creates a new object assign id of the object as id=(node_id, ++last_obj_id). And F is the Set of functions that can be applied to the object. Any f ∈ F, has unique function code relative to object, and takes in arguments array args. Each object has a special function delete() associated used to delete the object. Corresponding to each f ∈ F, their is a unique function f such that if o.f(args) = o , then o.f(args) = o, i.e. fof = identity function.

Function create(st, F) is used to add new object (generated unique id, state given by parameter st, functions set given by parameter F) to add new objects to the shared context.

Events: An Event is a 5-tuple (f, o, t, node, s) where:

f = function to be applied on object with id o (can be stored as 2-tuple (function_opcode,args))
t = logical timestamp of event
node = sender
s = sequence number of event

Function execute(Event e) is used to execute the function Obj(e.o).f(e.f.args)) and returns the updated object.
Function execute_undo(Event e) is used to execute the function Obj(e.o).f(e.f.args) and returns the updated object.

Total Ordering for Concurrency Control

Total ordering enforces all nodes to execute events in the same order, this can be achieved if events are delivered in casual atomic order, this method however will lead to slow response time in the system since operations cannot be executed immediately at local site.

Another way to achieve this is by using CBCAST[3] with additional undo and redo operations to enforce atomicity. When two concurrent events are received at a node, their priority order is determined by id of sender associated with each event. All concurrent events with lower priority(sender id) already delivered to a node are first undone in reverse sequence, received event is delivered and undone events are redone in sequence.

Based on Lamport's Logical Ordering of Events, we extend Lamport's "happened before(<)" relationship to $<_e$ to compare two concurrent events. For two events, e and f, e $<_e$ f , if (e.t < f.t or e.node < f.node)

```
SendEvent(Event e){
    t[me] = t[me]+1;
    seq += 1;
    lastSeq[me] = e.seq = seq;
    lastClock[me] = t;
    e.t = t;
    e.node = me;
    execute(e);
    L.push(e);
```

```
        Multicast(e, S - {me});
}

ReceiveEvent(Event e){
        Q.add(e);
        E = [] //Queue of events to execute, (same data structure as of Q)
        U = [] // Stack of events to undo
        for (e in Q){ //traversing from oldest to most recent(logically) event in queue
                if(lastSeq[e.node] = e.s-1){
                        if( e.t[n] <= t[n] for all n  S - {me} ) {
                                E.push(e);
                                lastSeq[e.node] = e.s;
                                t[e.node] = e.t[e.node];
                                lastClock[e.node] = e.t;
                                t[me] = t[me]+1;
                                for (node in S) t[node] = max(t[node] , e.t[node]);
                        }
                }
        }
        f = L.last();
        if(E.last().t $<_e$ f.t){
                E.push(L.pop());
        }
        for each (e in U){
                execute_undo(e);
        }
        for each (e in Q){
                execute(e);
        }

            //discarding unrequired log
        for(e in L){
                if(e.t <= Min(lastClock)){
                        L.remove(e);
                }
        }
        if(lastClock[me] < t){
                SendEvent(AliveEvent);
        }
}
```

Bounding Size of Log

In above discussed algorithm, each node needs to maintain a log of events in the order of execution, so that they can be undone when an older event is received. This requires that space required by algorithm increases linearly with number of events executed. Hence, we need a procedure to discard old event logs which are no longer necessary. For this, we discard events in L whose logical clock is such that no older event will ever arrive. If a site is not sending any events i.e. is idle, then size of log will never decrease, hence we use still alive events. The still alive events can also be monitored by the system to detect sites that have crashed or are unreachable.

Total Ordering vs Partial Ordering

In the above discussed algorithm partial ordering may initially get executed at a site, but when an older event arrives, corresponding undos and redos are done to ensure total ordering of events delivered at quiescent.

However, goal of a groupware application is not to enforce total ordering explicitly but to preserve precedence and convergence properties. These properties can be conserved even if total ordering is not enforced explicitly but is implied through a partial ordering, i.e. if a partial ordering can bring the system in same state as total ordering then a node can execute that partial ordering in order to preserve convergence property.

Independent Objects and Properties of Events

Two objects are said to be independent from each other if any operation on one object does not affect the other, i.e. , for all events e1 and e2 independently modify objects o1 and o2 to o1 and o2 respectively, then execution sequence e1,e2 and e2,e1 results in same updated objects o1 and o2.

Two events are commutative if they result in same final state of the shared context if performed in any order. An event e1 masked by e2 if the execution sequence e1,e2 and e2 results in same final state of the context. Two events are in conflict if they neither commute nor mask.

```
commute(Event e1, Event e2){
    if(e1.o == e2.o){
        return Obj(e1.o).(e1.f)(e1.args).(e2.f)(e2.args) == Obj(e1.o).(e2.f)(e2.args).(e1.f)(e1.args);
    }else{
        if(e1.o and e2.o are independent objects) return true;
        else return false;
    }
}

//returns if e1 masks e2
mask(Event e1, Event e2){
    if(e1.o == e2.o){
        return ( Obj(e1.o).(e2.f)(e2.args).(e1.f)(e1.args) == Obj(e1.o).(e1.f)(e1.args) );
    }
    Return false;
}

commuteWithSet(Event e, Event Array S){
    for(f in S){
        if( commute(e,f) == false){
            return false;
        }
    }
    return true;
}
```

ORESTE Concurrency Control Algorithm

This algorithm relaxes total ordering of events by allowing execution of an event out of order if it results in same final state as total order. When an event older than those which are executed is received, promote procedure is called on the event. This procedure calculates all the events to be undone i.e. events which does not commute with arrived event. If any of the newer event in the Log masks arrived event then it is not required to execute the arrived event at all. The algorithm executes all operations on an object as they arrive (if events does not arrive in causal order then also they are executed and when older event arrive required undos are calculated by promote procedure.) if the object creation has been executed otherwise the events are buffered till object creation event arrives.

```
sendEvent(Event e){
    t[me] = t[me]+1;
    seq += 1;
    lastSeq[me] = e.seq = seq;
    lastClock[me] = t;
    e.t = t;
    e.node = me;
    execute(e);
    L.push(e);
    Multicast(e, S - {me});
}

receiveEvent(Event e){
    o = Obj(e.o);
    if(o == null){
        if(f == create){
            execute(e);
            L.push(e);
            for(e in Q){
                if(e.o == o){
                    execute(e);
                    Q.remove(e);
                }
            }
        }else{
            Q.push(e);
            return;
        }
    }else if(o == deleted){
        L.push(e);
    }else{
        if(t $<_e$ e.t){
            execute(e);
            L.push(e);
        }else{
            Promote(e);
        }
    }
    t[me]=t[me]+1;
    for (node in S) t[node] = max(t[node] , e.t[node]);
    if(e.s == lastSeq[e.node] + 1){
        lastSeq[e.node] = e.s;
        lastClock[e.node] = e.t;
    }
    //discarding unrequired log
    for(e in L){
        if(e.t <= Min(lastClock)){
            L.remove(e);
        }
    }
    if(lastClock[me] < t){
        SendEvent(AliveEvent);
    }
}

// We assume that Log L always keeps itself sorted by Logical Timestamp

Promote(Event e){
    U = []; //stack of events to undo-redo
    while(U does not change){
        for (f in L where e.t $<_e$ f.t){
            if(commuteWithSet(f,U)){
                if(mask(f,e)){
                    Return; //masked event is not executed
                }else{
                    if(not commute(f,e) ){
                        U.push(f);
                    }
                }
            }
        }
    for (e in U.reverse()) execute_undo(e);
    execute(e);
    L.push(e);
    for (e in U) execute(e);
}
```

Commute and Mask

Commute and Mask functions described above are application specific in terms of implementation. They can be define commute and mask in terms of operations that can be performed on an object. For instance, change in position and color of same circle object are commutative, if all objects are independent, then the two operations will always be commutative no matter on which object they are called on.

Instead of classifying functions as whether they are commutative or not, we can access objects in hierarchy and call same update function to change their value. Accessing independent objects this way will dispose the need to classify multiple operations as commutative. Another advantage of accessing objects in hierarchy is that for two update calls on same object, older one will always be masked by the newer request.

For some objects (which can be deleted and created via user interface) we can use delete and create operation, operations assigned to an object which is not yet created are deferred till it is created, and delete operation always masks other operations on the same object.

Collaborative Flowchart Drawing Tool

In this section we will describe a collaborative flowchart drawing tool based on model explained above. Users have canvas on which they draw, the canvas is not created explicitly in objects, but it is the shared context which is accessible to users via GUI. A user can draw different shapes(ellipse, rectangle, line etc) and textareas. All objects have properties size(width, height), position(x,y,z-index), fill color. Textareas have addition properties text and textcolor.

Structures:
Size: 2-tuple (int w, int h)
Position: 3-tuple (int x, int y, int z)
Color: 3-tuple (char r, char g, char b)
Info: 5 tuple of basic properties of each object, (Size size, Position position, Color fillcolor)
Circle: (Info info)
Rectangle: (Info info, Color textcolor, String text)
Line: (Info info)

Operations on Structures:

The operations mentioned here are those which are sent by nodes to each other, they do not represent how application at each node process user inputs.

update: This operation can be called on any data type with all updated properties of the data type passed as arguments.
create: This operation can be called to create Circle, Rectangle and Line objects with all properties as argument.

delete(): This operation can be called on Circle, Rectangle and Line objects to delete them.

Note that any update on one object can not affect another object. Hence all objects are independent of each other with respect to update, create and delete operations. Also note that older update operation on same object is masked by a younger one. As mentioned in Oreste algorithm, all operations on an object are deferred till object creation is received. All operations after deletion of an object are ignored and simply pushed to log. For this application commute and mask can be defined as follows:

```
O: Set of all objects (shared context)
commute(Event e1, Event e2){
    if(e1.o == e2.o){
        return false;
    }else{
        return true;
    }
}

//return if e1 mask e2 given e2 happened before e1
mask(Event e1, Event e2){
    if(e1.o == e2.o){
        if(e1.f == delete) return true;
        else if(e1.f == update and e2.f == update) return true;
        return false;
    }
return false;
}
```

The reason this definition of commute (commutative if object ids are different else not) work is because of two reasons, one is that all objects in application are independent. Other reason is that objects are accessed in hierarchical order. For eg. if a Circle object, say c1, is moved and color is changed by two different nodes concurrently, then the events are commutative, the corresponding events sent would be (update, c1.info.position, <timestamp>, <node id>, <seq no>) and (update, c1.info.color, <timestamp>, <node id>, <seq no>). Although user modifies same object but the update operation is performed on different objects which are part of the same object which user modified via interface.

Group Chat Application

Consider a simple group chat application in which each user can send a text message to other users. In this context, we can think of the chat window as the shared context and the messages being sent as the objects within it. The only action user can perform is send a message which creates the objects. Therefore we have only one Object Type and only one operation.

Object Types:
Message: (String msg)

Operations:
create(String msg, int seq): This operation simply creates the message object delivered in chat window. Int seq is the sequence number of the message in the chat window from bottom, therefore whenever a user will send a message the application will always set seq to 0 (bottom most message), so their is no need to send this parameter with events.

For this example mask and commute always returns false.

Dependent Objects

One can see that all objects are not independent of each other in this application. Although all nodes only send message string in events but another property associated with each message is the relative position where they are displayed.

All messages that were delivered before an older message arrived need to shifted in the interface. With respect to Promote procedure of ORESTE algorithm all message events delivered before delivery of an older message are dependent on this older event and will be undone by Promote algorithm. This is not a very elegant solution since the message can be inserted at its correct position without undoing and redoing creation of all other newer message objects. Instead of undoing creation of newer messages we can modify the seq parameter of create request from 0 to the correct position in chatbox. This idea is based on the Distributed Operational Transform (dOPT)[4] Algorithm, key idea is to modify received operation such that final state is same as if the operations were carried in total ordering.

Modified ORESTE Algorithm

In this section we present a modified version of ORESTE Algorithms Promote procedure to include operation transform to further reduce undo and redo operations.

```
Promote(Event e){
  U = []; //stack of events to undo-redo
  while(U does not change){
        for (f in L where e.t $<_e$ f.t){
              if(commuteWithSet(f,U)){
                    if(mask(f,e)){
                          return; //masked event is not executed
                    }else{
                          if(not commute(f,e) ){
                          //find e such that execution sequence f e and e f gives same final state
                          e = transform(e, f);
                          if(e == none) U.push(f);
                          else e = e;
                    }
              }
        }
  }
  for (e in U.reverse()) execute_undo(e);
  execute(e);
```

```
        L.push(e);
        for (e in U) execute(e);
}
```

transform(Event e, Event f); function finds Event e such that execution sequence e,f is equivalent to f,e. For the given algorithm to work the function must satisfy additional constraint that Event j, commute(j,e) commute(j,e), without this constraint algorithm will have to restart the Promote procedure for transformed Event every time we transform received event.
Transform function for Group Chat Application described above is defined here:

```
transform (Event e, Event f){
        e = e;
        e.f.args[1] += 1;
        return e;
}
```

Collaborative Computing Transport Layer

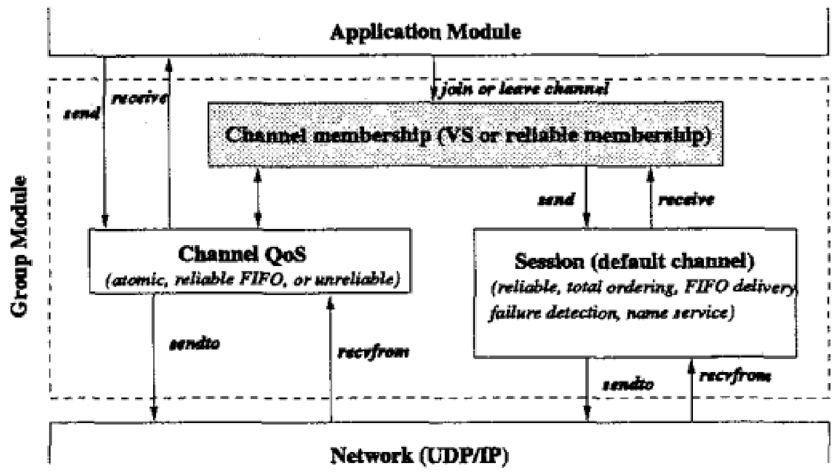
Architecture

The word group is commonly used in distributed environments for sharing information among related process, and multicasting makes information known to all group members. In CCTL light-weight groups (LWGs) are called channels, whereas heavy-weight groups (HWGs) are called sessions. Related channels combine to form sessions. Session and channels both have the same basic operations (join, leave, send and recieve). Quality of service of a particular channel is fixed at the time of creation whereas the session memebers may join and leave channels dynamically. As soon as the last participant leaves both channels and sessions are destroyed. A sessions provides a default, virtually synchronous reliable multicast service among participants that is used to implement group management services.

CCTL is basicly implemented as a group module interposed between application and physical network. The group module consist of channel membership (CM), communication (COM), and session sub modules. The CM module implements the membership protocol for channels. The COM module implements data communication service of channels. The session module supports the service that session provides.

While implementing membership protocol CM module uses session module to communicate with the CM module of other session members, and uses COM module enforce vitual synchrony for the channels. If a channel requires virtual synchrony, the application module uses the CM module to send and recieve messages, and if it does not then the messages are send and recieved directly through COM module. The session module the CM module of session membership updates including a member failures. A process must join a session to participate in a collaboration session. A CCTL channel currently supports one of three data communication services: total ordering reliable, FIFO reliable and unreliable.

Figure 1: CCTL architecture[5]


Application Module
send
receive
join or leave channel
Channel membership (VS or reliable membership)
send
receive
Group Module
Channel QoS
(atomic, reliable FIFO, or unreliable)
Session (default channel)
(reliable, total ordering, FIFO delivery
failure detection, name service)
sendto
recvfrom
sendto
recvfrom
Network (UDP/IP)

Protocols

- Membership for unreliable channels

  Unreliable channels are useful to applications such as audio and video which require timely delivery but is not much affected by some message loss. Requiring two processes to receive the same messages in the same view is meaningless if messages may be lost. Unreliable channels provide a minimal guarantee that a member change is recognized by every channel member. A process p wishing to join an unreliable channels calls the channel join which sends a join message through the session module. The channel module recieves it join message totally ordered with respect tot other membership messages as the message is rebroadcasted by the session owner. The channel is created after recieving channel join. Since all channel updates are totally oredered, no two process choose different channel addresses for the same channel. The joining process gets the multicast address of the channel for data communication. The leave operation is also done by simply multicasting a leave channel message through the session module. In case of no member for a channel, the CM module removes the channel and deallocate the multicast address.

- Membership for atomic channels

  Process p multicasts a join view change request through the session module. On recieving the join request the oldest channel member q sends a view change acknowledgment (VCA) to all non faulty channel members on the same channel. The channel being joing totally orders every message therefore this acknowledgment will be totally ordered. If the channel p wishes to join contains no members, then we make p the creator. If p is the creator, the join view terminates and the channel view contains only p. Else p waits for a VCA message from the leader, containing the correct membership. On recieving VCA all channel members update their views and deliver the join. For leave, p is removed from the channel on reciept of VCA. To enforce virtual synchrony, all processes must recieve the same messages in a given view. To guarantee virtual synchrony, the sender p comares the sending and recieving views for each messages sent. Process p retransmits messages to the process that are in the recieving view of the messages but not in the sending view of the messages.

Figure 2: Dotted arrows - atomic session multicast, solid arrows - atomic channel multicast. [5]

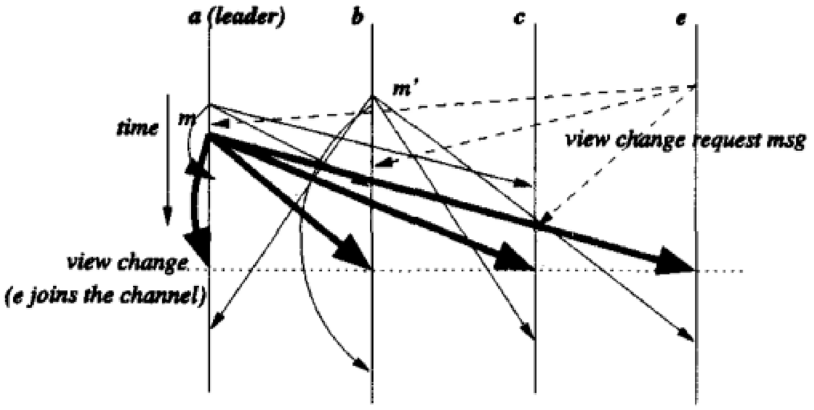
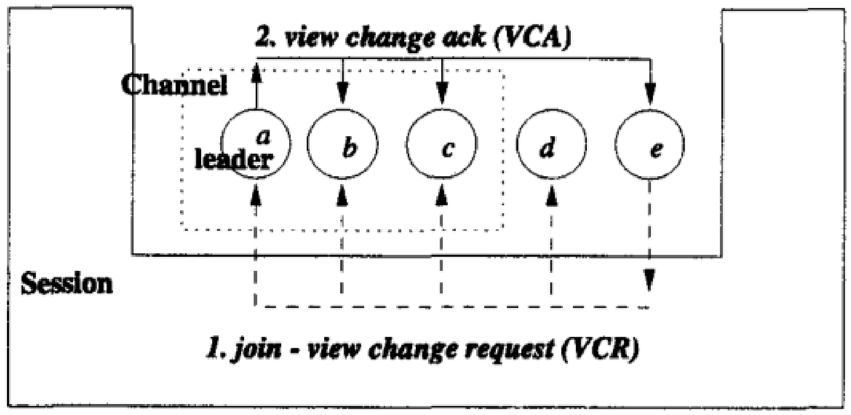


Figure 3: Regular messages m and m ordered with respect to a view change message (thick arrows). [5]



- Membership for reliable channels

  To join or leave process p sends VCR to the default session channel, then all channels sends VCA on the reliable channel. On recieving VCA, until ps view change request is not acknowledged by all non faulty channel members the subsequent messages from r are delayed by processes. As soon as all channel members acknowledges, a process adds p to its view, deliver join and process delayed messages. View changes and regular messages should be ordered on reliable channels as atomic cast is not available to us. Since the VCR are totally ordered and the channel implements a reliable (FIFO) multicast, every message that a process sends after its transmission of VCA, is delivered after the corresponding view change. Thus the protocol guarantees that within the same view, all non faulty members will recieve the same set of messages. View changes need n+1 multicast where n is the total number of channel menbers. We can also optimize this ny simply having the leader collect and send the index numbers of last message each channel member sent. Now the channel members can delay the delay the delivery of view change messages until all messages with smaller index number are recieved from all processe. As soon as the earlier messages are recieved, process deliver the join or leave and continue processing delayed messages.

Audio Conferencing

CCFAudio is an Internet based audio conferencing tool with the features to support its use in a collaborative session. Achieving the audio quality comparable to internet phones, which are point to point and with no shared resources, is a difficult task. CCFAudio is a combination of three subsystems : send subsystem , receive subsystem and GUI subsystem. Send subsystem packetizes the audio signals and transmits them to other tools. Receive subsystem receives those audio packets , mix them and play the result. GUI subsystem controls interface part with click and playback buttons along with the graphical part showing the present participants and marking the speaker. User-specific parameters of the CCFAudio is also managed by the GUI subsystem. After initial setup of these parameters, operation of CCFAudio is completely hands-free.

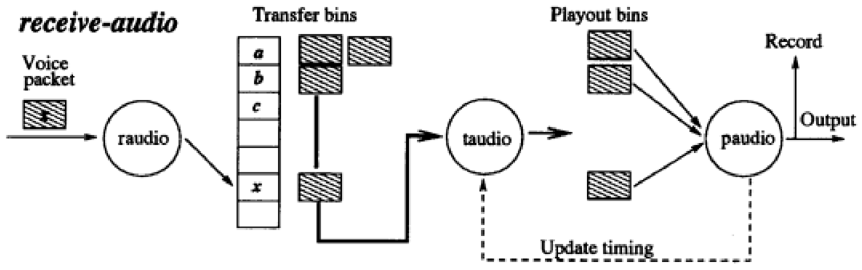
Send-Audio subsystem

There are two threads saudio and faudio. The sources for the threads are an audio input device(e.g. microphone) and an audio data file respectively. The saudio thread repeatedly samples a fixed amount of audio data taken through the device like microphone and stores it into a circular buffer and sample is then checked for the voice. If the voice amplitude exceeds certain threshold *v* then it confirms the presence of a voice. The faudio thread takes an input audio data file, read and encode it and transmits the data at the rate given in file header.

Receive-Audio subsystem

It consists of three threads i.e, raudio(receives) , taudio(transmits) and paudio(plays). *raudio* receives the audio packets from all the other users and deposits them to respective transfer bins, one bin per collaborator. Samples from a transfer bin are marked playable when the number of samples exceeds a given playout threshold *t* or "End-Of-Speech-Spurt" indication is received. *taudio* periodically removes the playable samples from each of the bins and transfer it to playout bins. *paudio* mixes the sample in the playout bins and send the result to audio output device. It also computes the time t_{end} when the audio device becomes idle, the thread updates the time to the sum of previous idle time and time needed to play out the new data if the current idle time exceeds the previous one. Else, it updates to the sum current idle time and time needed to play the data. The t_{end} value is fed back to *taudio* thread, which uses it to compute the deadline to wait for late packets.

Figure 4: Figure of Receive-Audio System[6]



Frequent interruption in data stream results in inconvenient conversation. It is therefore important to ensure that system does not get overloaded dut to overflow or idle due to underflow. Overflow is a result of fast data transfer to output audio device than it can drain, whereas underflow occurs due to gaps in the stream. Arriving packets can be re-ordered if subsequent packet has not been played. In case of round trip delay exceeding a certain value , audio packets needs to be played as soon as possible to avoid overflow or underflow.

GUI subsystem

It manages the graphical interface to control the click and playback options. It also shows the list of present participants and person speaking at the moment and works to invite other participants. The parameters *k1*, *k2* and *v* are managed through GUI only. The parametric values will be saved in a file at the end of the session, to be further used as it is.

References

[1] Leslie Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM, v.21 n.7, p.558-565, July 1978

[2] A. Karsenty, M. Beaudouin-Lafon, ”An Algorithm for Distributed Groupware Applications”, Proc. Int’l Conf. Distributed Computing Systems, 1993

[3] K. Birman, A. Schiper, P. Stephenson, ”LightWeight Causal and Atomic Group Multicast”, ACM Trans. Computer Systems, vol. 9, no. 3, pp. 272-314, Aug. 1991

[4] C. Ellis, S.J. Gibbs, G. Rein, ”Concurrency Control in Groupware Systems”, Proc. ACM SIGMOD ’89 Conf. Management of Data, pp. 399-407, 1989

[5] I. Rhee et al., Group Communication Support for Distributed Multimedia and CSCW Systems, Proc. 17th Intl Conf. Distributed Computing Systems, IEEE Computer Soc. Press

[6] Sarah Chodrow, Michael Hircsh, Injong Rhee, and Shun Yan Cheung. Design and implementation of a multicast audio conferencing tool for a collaborative computing framework. In JCIS, March 1997

[7] Suman Banerjee , Bobby Bhattacharjee , Christopher Kommareddy, Scalable application layer multicast, Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications, August 19-23, 2002, Pittsburgh, Pennsylvania, USA