M2MC: Middleware for Many to Many Communication over broadcast networks

Chaitanya Krishna Bhavanasi and Sridhar Iyer
KR School of Information Technology,
Indian Institute of Technology - Bombay
(email:chaitanya, sri@it.iitb.ac.in)

***Abstract*— M2MC is a new distributed computing middleware designed to support collaborative applications running on devices connected by broadcast networks. Examples of such networks are wireless ad hoc networks of mobile computing devices, or wired devices connected by a local area network. M2MC is useful for building a broad range of multi-user applications like multiplayer games, conversations, group ware systems.**

Unlike existing middleware architectures that rely on central servers, M2MC is truly distributed protocol. Applications developed using M2MC do not require central servers for message ordering, member synchronization and group management. Being broadcast oriented, M2MC do not require any resource consuming routing protocols.

M2MC architecture consists of Message Ordering Protocol, Member Synchronization Protocol and protocols for processes to join and leave the groups. In this paper we describe key components of M2MC and their implementation in Java.

I. Introduction

Middleware is a set of software facilities that mediates between an applications programs and communication network. It manages the interaction between applications across the computing devices by providing communication support in a transparent way to the application.

The communication paradigms can be classified into one-to-one or one-to-many or many-to-many communications. As shown in Fig1a, in one-to-one communication model only two processes are involved in communication, one for sending the message and other receiving it. Examples of applications of one-to-one paradigms are web browsing, email etc. The Fig1b, shows one-to-many communication in which one process sends message and many processes receive it. Web casting is an example of such application. The Fig1c shows many-to-many communication paradigm in which a message sent by any of the processes reach every process present in the network and is interested in receiving it. The Fig1d shows mulitple group communication using many-to-many communication.

In this paper we propose M2MC, a middleware for supporting applications requiring group and many-to-many communication patterns. M2MC consists of a set of protocols for building collaborative applications like mulitplayer games, chat applications etc. that run in broadcast networks. Some examples of such networks are: wireless proximal ad hoc networks of fixed or mobile devices or hybrid network connecting wired and wireless devices. The Fig2 shows the relative position of M2MC protocols with respect to the application and network protocols. M2MC consists of Message Ordering Protocol (Section 4), Group Join /Leave Protocol (Section 5) and Member Synchronization Protocol (Section 6).

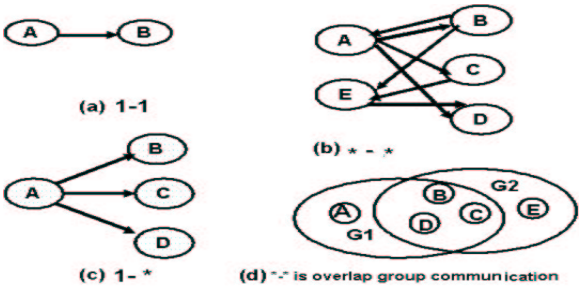

Fig. 1. Communication Paradigm

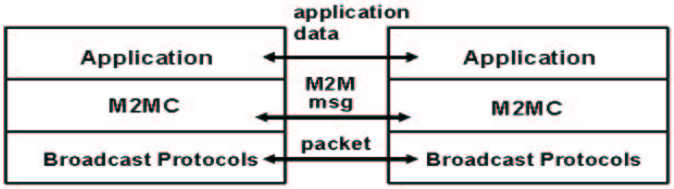

Fig. 2. M2MC

Unlike existing middleware architectures that rely on central servers, the M2MC is intended for running collaborative applications without relying on central servers. In wireless ad hoc network or hybrid network of devices, relying on central servers is not attractive because the devices are not necessarily always in the range of wireless access point. Furthermore relying on any one wireless device to act as a server is unattractive because devices may come and go without prior notification. Using M2MC, those devices which are in range of each other can act in conert to run the application. We have developed an application using our M2MC APIs called Threaded Chat application.

II. Motivation

In this section we describe the Threaded Chat application to motivate the relevance of M2MC. Consider a group of processes (A,B,C,D) running on distributed devices and implementing a simple chat application that lets the members of the group interact with each other. The processes communicate with each other by sending messages using a broadcast medium. Suppose the application is implemented using a

message ordering protocol based on logical timestamps, such as total ordering [2]. See [9] for a comprehensive survey of total ordering protocols.

As shown in Fig 3, let process C send messages *'Did you visit Delhi?'* and *'Did you visit Chennai?'* with timestamps 1 and 2 respectively. After receiving the above messages, suppose process A replies to the message *'Did you visit Chennai?'* with the response 'No' and process B replies to the message *'Did you visit Delhi?'* with the response 'Yes'. As per total ordering, both A and B would affix the timestamp 3 to their responses. Now, the message ordering protocol at process D on receiving these messages orders them according to their timestamps and displays them on the chat console. However, since there are two messages having the same timestamp, they may get displayed on the console at D in an arbitrary order. This leads to ambiguity because the user at D may not be able to map the responses 'No', 'Yes', to the messages *'Did you visit Delhi?'*, *'Did you visit Chennai?'* appropriately. Hence total ordering protocol is inadequate for such an application. It can be shown that the ambiguity persists even when the messages are ordered using vector clocks, as in causal ordering [6] or even when synchronized global clocks [5] are assumed.

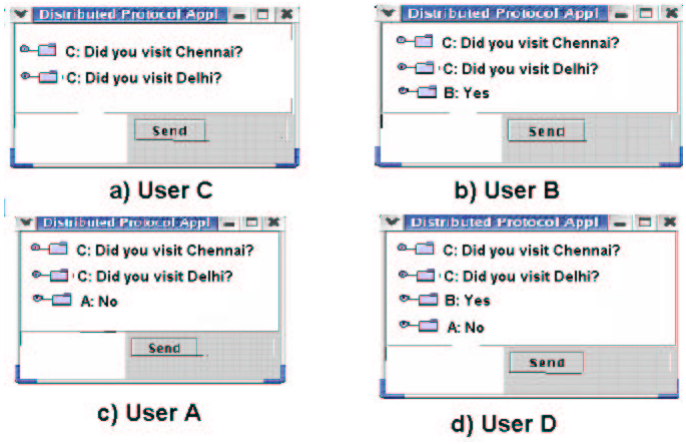

Fig. 3. Chat Application

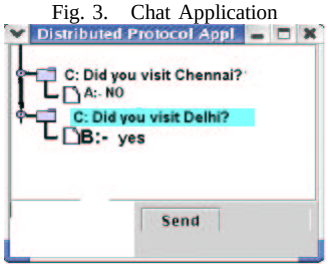

Fig. 4. Threaded Chat Application

In contrast to the above, consider a Threaded Chat application [7] that lets users communicate in a *message-response* form as shown in Fig 4. All chat messages are structured in the form of a tree. The key feature of this tree structure is that messages and responses are organized into relationships called **threads**. A user explicitly selects a message before responding to it. As a result, the response is linked directly to the corresponding message, using threads, and other users can perceive the semantic relationship among the messages.

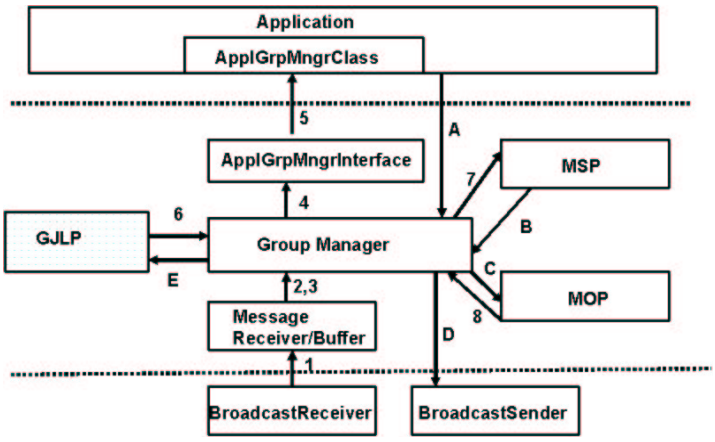

Fig. 5. M2MC architecture

Although the paper [7] does not provide the details of the Message Ordering Protocol used by Threaded Chat application, such an application can be easily implemented using the Message Ordering protocol of M2MC. For the above example, upon receipt of messages from process C, process A displays both of them to the user. Now the user at process A would explicitly select the message *'Did you visit Chennai?'* before responding with the message *'No'*. The Message Ordering Protocol at process A captures this semantic dependence between the message and its response and sends this information to the group, along with the response. Similarly, the Message Ordering Protocol at process B captures the semantic dependence between the message *'Did you visit Delhi?'* and its response *'Yes'* and sends this information to the group. The Message Ordering Protocol at D, upon receipt of these responses, orders the messages appropriately and unambiguously, as shown in Fig 4. If a new process E enters the broadcast domain then it executes the Group Join and Leave protocol to get the list of group of applications running in the broadcast domain. After joining the group, the Member Synchronization Protocol will ensure that the process E receives all the messages that have been sent to the group. The Message Ordering Protocol orders these messages and displays on the screen.

In the next section, we describe the architecture of M2MC and its various components.

III. Middleware Architecture

As shown in Fig 5 M2MC comprises of a Message Ordering Protocol (MOP), Member Synchronization Protocol (MSP), and protocols for process to join and leave the group called Group Join Leave Protocol (GJLP).

A. Components of Middleware

The following are components of our middleware.

*1) **Message Ordering protocol (MOP)**:* When two or more messages are sent to the group, processes receive them in arbitrary order depending on the transmission delays between

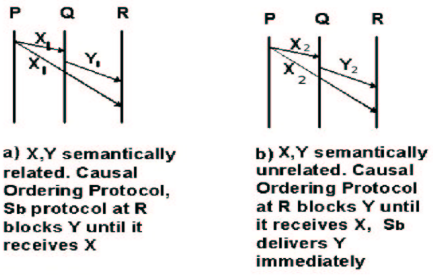

Fig. 6. Illustrating S_b ordering

senders and receivers. For example, if a group member sends a message Y_1 as a response to message X_1, then it is possible that some of the group members may receive Y_1 before X_1 (see Fig 6). Hence an ordering protocol is required to guarantee that every group member will deliver the message X_1 before delivering Y_1 to the application. Also if X_2 and Y_2 are any two *semantically unrelated* messages, then a member receiving Y_2 before X_2, should not block delivery of Y_2 by waiting for the arrival of X_2.

Traditional solutions like total ordering protocol [6] or causal ordering protocol [2] do not take into account the semantic relationship among messages and hence are inadequate for many distributed group communication applications.

For M2MC, we propose a new message ordering, called S_b ordering, and a corresponding protocol, called Message Ordering Protocol (MOP) , which is implemented by every member of the group. The primary objective of the MOP is to order received messages, based on the semantic relationship among them, irrespective of the chronological order in which they are received. As a result, the MOP also minimizes the *delivery delay* at a process (the time from the moment a message is received at a process to the time the message is delivered to the application consuming it), by blocking delivery of a message only if it is yet to receive any semantically preceding message(s).

2) ***Group Join Leave protocols (GJLP)****:* When a process is newly connected to a broadcast network or a process leaves the network temporarily and later rejoins, it should be aware of the group applications that are running across the network so that it can join in any of the interested applications. For M2MC, we propose Group join protocol to keep a the newly connected process aware of group applications running across the network so that it can join them and a Group Leave Protocol for a process leaving a group application, to informs its departure to the members of the group. The protocol is also for creating new groups.

3) ***Member Synchronization Protocol (MSP)****:* When a process newly joins a group or it leaves an existing group and later rejoins it, it misses messages that were sent to the group during its absence. The process must recover such lost messages, as soon as it joins the group, such that the group applications continue running correctly. For M2MC, we propose Member Synchronization Protocol to recover such lost messages.

4) ***Group Manager****:* The Group Manager regulates the flow of messages from one component to another. It receives messages from the network, determines the nature of the message, routes them to the other components of M2MC appropriately and finally delivers it to the application. It also receives messages from application and delivers it to the network layer for broadcasting. Since M2MC supports applications over multiple overlapping groups, the group manager maps group information to the corresponding instances of the MSP, MOP allocated to the groups.

5) ***Broadcast Layer****:* The broadcast layer is assumed to be reliable and guarantees message delivery to every member of the group. However it may suffer from nondeterministic bounded delay in message delivery. Messages in transit need not follow FIFO order. The Broadcast layer supports functions for broadcasting a message to the group and for receiving a message from the group.

6) ***Applications****:* The middleware supports group applications like chat applications, multiplayer games, group ware systems etc. The application uses the APIs provided by GroupManager and ApplGrpMngrInterface for sending and receiving group messages, for creating new groups or for joining and leaving existing groups.

B. Middleware operations

1) ***For creating a new group:****:* The Application calls (step A in Fig 5) GroupManager component with group description parameters. The GroupManager creates an identity and instances of various protocols for the new group and broadcasts the new group description(step D).

2) ***For joining an existing group :****:* Every process maintains a *groupsInfoList* containing the identity and members information of every group that it is aware of. When a process newly connects to broadcast network or a process leaves the network temporarily and later reconnects, it broadcasts its presence by doing the following. The GroupManager calls GJLP protocol component, gets the advertisement message (steps E, 6) containing the attributes of process and broadcasts by sending it to broadcast layer along step D. Every process (including the one that sent the advertisement) on receiving advertisement (along steps 1, 2, 3) gives it to their respective GroupManager. The GroupManager calls GJLP and gets (steps E, 6) the *groupsInfoList* containing identities and members list of every group that the process is aware of. The GroupManager broadcasts the *groupsInfoList* by sending it to broadcast layer along step D.

At every process, *groupsInfoList* received by the broadcast layer reaches GJLP along 1, 2, 3, E. The GJLP checks if there exists information about any new group in the received message that is not present in its *groupsInfoList*. If such group exists then it presents the details of the new group to the application along 6,4,5. The user at the application if decides to join a group, calls GroupManager (step A) which in turn creates instances of MSP, MOP for the group and sends **joinMsg**, containing the identity of the group and

identity of the process, to the broadcast layer (along step D) for broadcasting. Every member of the group on receiving the **joinMsg** updates their **groupsInfoList** by appending the identity of the process specified in **joinMsg** to the members list of the group.

For leaving the group, application calls GJLP along step A, E. GJLP creates a **leaveMsg** with identities of the process and the group that the process is leaving. The **leaveMsg** reaches GroupManager along 6 and it subsequently broadcasts along E. The GJLP at every member of the group on receiving **leaveMsg** along 1,2,3,E updates their **groupsInfoList** by deleting the identity of the process from the members list of the group specified in **leaveMsg** message.

*3) **For sending a message to group**:* For sending an application message **M** to the group, the application calls (arrow A) GroupManager which in turn calls MOP. The MOP adds appropriate headers and returns the message (arrows C, 8) to GroupManager. The GroupManager broadcasts the message to the group by calling Broadcast layer.

*4) **On receiving a message from the broadcast layer**:* The GroupManager on receiving the application message *M* (along arrows 1,2,3) finds the group that the message belongs to and calls MSP of the group (along 7,B) for storing the message. It then calls MOP (along C) of the group. MOP checks whether the process received all the messages that are semantically before *M*. If it has received them then it sends (along 8) *M* and any other messages that are waiting for its arrival (because *M* is semantically before waiting messages) to the GroupManager which in turn delivers them to the application (along 4). Otherwise MOP blocks the delivery of the message.

*5) **Member synchronization**:* When a process running on a mobile device leaves the network temporarily and later reconnects to the network it misses the messages that were sent to the group. It executes Member Synchronization Protocols for each group (in which it has membership) for recovering the missed messages.

The GroupManager calls MSP and gets (arrows 7, B) a synchronization request message *SyncReqMsg*. *SyncReqMsg* contains the process identity, group identity, and application messages the processes has received before leaving the network. The GroupManger broadcasts it by sending it to broadcast layer (along D).

BroadcastReceiver on receiving *SyncReqMsg*, sends it to GroupManager (steps 1,2,3). The GroupManager at every process other than the process that sent SyncReqMsg sends it to MSP (step 4). MSP finds for any messages that it has not received from the group, in the application messages of *SyncReqMsg* and sends them to the Group Manager (step B). The Group Manager subsequently delivers them to the application. MSP creates a temporary *applMsgList* containing application messages that it has received from the group. It waits for random time and sends **SyncRespMsg** containing the messages present in *applMsgList*. While waiting, if MSP receives *SyncRespMsg* sent by some other process, then it deletes the messages from *applMsgList* that are present in *SyncRespMsg*.

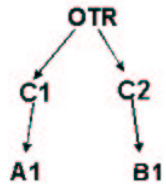

Fig. 7. Ordering Tree

The broadcast layer at the process that sent *SyncReqMsg* on receiving *SyncRespMsg*, sends it to GroupManager (steps 1,2,3). The GroupManager subsequently delivers the application messages present in *SyncRespMsg* after sending to MSP,MOP for storing and ordering purposes.

Hence the process that missed the messages gets updated with the application messages and also every other process gets the application messages from the process that requested synchronization.

IV. Message Ordering Protocol

In this section we present the specification and implementation details of Message Ordering Protocol. Group Join/Leave Protocol and Member Synchronization Protocols are presented in next sections.

The primary objective of the Message Ordering Protocol called MOP is to identify the semantic relationship among received messages and delivering them to the application in a semantically consistent order. Guaranteeing such ordering involves:

1) Capturing the semantic relationship between a message and its response, from the application at the sender.
2) Representing these semantic relationships in an appropriate form and conveying them to the receivers.
3) Maintaining the relationship at each member of the group with minimum overhead.

A. S_b Ordering

We represent the semantic relationship among the messages using S_b order relationship defined as follows:

Two messages X and Y are said to be related in S_b order if and only if Y is produced semantically in response to a unique message X. This is represented as $X \xrightarrow{S_b} Y$. Also if X and Y are not semantically related then it is represented as $X \xrightarrow{\neg S_b} Y$.

For a group of messages, we conceptually represent the semantic relationship among them in the form of a tree, called the **Ordering Tree** (OT), as shown in Fig 7. The OT has the following structure:

- The vertices of the OT are identities of the messages; each message has a unique system-wide identity.
- The directed edges of the OT represent the semantic *message-response* relationships among messages. There is an edge between any two vertices in the OT, if and

only if and the corresponding messages are related in S_b order.

- The root of the OT is a virtual node, denoted by OTR. OTR is assumed to be semantically before all the messages sent to the group. If a message is not a response to any other message in the OT, it is considered to be a response to OTR.

B. *Properties of S_b order*

Some salient properties of S_b order are as follows:

1) **Response semantics :**
   If X$\xrightarrow{S_b}$Y then P(Y) = X, i.e., X is said to be parent of Y.
   The OT represents this relationship in the form of a directed edge between a parent node X and a child node Y. For example in the Fig 7, node A1 is the root of the tree. A4 is produced in response to C1 (C1$\xrightarrow{S_b}$A4) and P(A4) = C1. Hence there is a directed edge from node C1 to node A4 in the OT.
2) **Uniqueness:**
   If X$\xrightarrow{S_b}$Y then P(Y) $\neq$ Z ($\forall$ Z, Z $\neq$ X), i.e., X is unique. The OT represents this by allowing a node to have multiple number of child nodes but a child node can have exactly one parent node. In other words, a message sent to the group may generate multiple responses from various members of the group but any given response is associated with one and only one message and not with multiple messages.
3) **Transitivity:**
   X$\xrightarrow{S_b}$Y $\bigwedge$ Y$\xrightarrow{S_b}$Z $\Longrightarrow$ X$\xrightarrow[*]{S_b}$Z.
   The OT represents this as having a path from X to Z, if there is an edge from X to Y and an edge from Y to Z. We use the notation $\xrightarrow[*]{S_b}$ to represent such transitive closure. It can be easily seen that the following also hold:
   - X$\xrightarrow[*]{S_b}$Y $\bigwedge$ Y$\xrightarrow{S_b}$Z $\Longrightarrow$ X$\xrightarrow[*]{S_b}$Z
   - X$\xrightarrow{S_b}$Y $\bigwedge$ Y$\xrightarrow[*]{S_b}$Z $\Longrightarrow$ X$\xrightarrow[*]{S_b}$Z
   - X$\xrightarrow[*]{S_b}$Y $\bigwedge$ Y$\xrightarrow[*]{S_b}$Z $\Longrightarrow$ X$\xrightarrow[*]{S_b}$Z

C. *Message Ordering Protocol*

Here we present the Message Ordering Protocol that includes:

1) **At the sender:** Captures the S_b order between a message and its response and includes this information while broadcasting the response.
2) **At the receiver:** Maintains the S_b order information and determines the action to be taken for each received message. A message is delivered immediately to the application either if its parent in the S_b order has been delivered or if it is not a response to any other message, i.e., it has the root of the OT (OTR)as its parent. Otherwise the delivery of the message is deferred, until the receipt and delivery of its parent.

We now describe the data structures and state diagram of MOP.

*1) **Notations, Message Format and Data Structures**:*

1) **Notations:**

   - $< gid >$: denotes the unique identity of the group.
   - $< pid >$: denotes the unique identity of the process in the broadcast network.
   - $< seqno_i >$: denotes sequence counter value at process **i**.
   - $< mid_i >$: denotes message identity of message i and $< mid_i >$ is $< pid_i, seqno_i >$
2) **Message Format:**
   A message format is: $< mid_c, mid_p, gid, data >$ where mid_c is the message identity, mid_p is the identity of its parent ($mid_p \xrightarrow{S_b} mid_c$), gid is the identity of the group and $data$ is the application information. If a message (mid_c) is not a response to any other message then the identity of its parent (mid_p) is set to OTR.
3) **Data Structures :**
   Every process maintains two data structures for every group:
   a) **Ordering Tree (OT):** As discussed earlier, the OT represents the S_b order among the messages of a group. Each process constructs its OT dynamically by recording the identities of those messages that have been received in S_b order.
   b) **Out of Sequence Messages Store (OSMS):** OSMS saves messages that have arrived out of S_b order. For every such message in the form $< mid_c, mid_p, gid, data >$, mid_c, mid_p, $data$ are are saved in the OSMS in the format $< Msg >$:$< mid_c, mid_p, data >$.

*2) **Protocol Actions ::*** The state diagram of the MOP is as shown in Fig 8. In the INITIAL state all the data structures are initialized to **NULL** and the process then waits in the LISTEN state. When the application wants to send a message to the group, the process goes to the RESPOND state, where it augments the message with the S_b order information, broadcasts the message and returns to the LISTEN state.

When a message is received from the group, the process goes to the RECEIVE state, where it checks the S_b order information of the message with the OT (Ordering Tree). If it has delivered the parent of the current message, it goes to the RCVDeliverableMSG state, else it goes to the RCVOutSequenceMSG state. In the RCVOutSequenceMSG state, the process simply saves the message in the OSMS and returns to the LISTEN state. In the RCVDeliverableMSG state, the process delivers the message to the application as well as any of its S_b order children that may be saved in the OSMS and returns to the LISTEN state.

A more detailed description of the protocol actions in each state, for a group of n processes, is as follows:

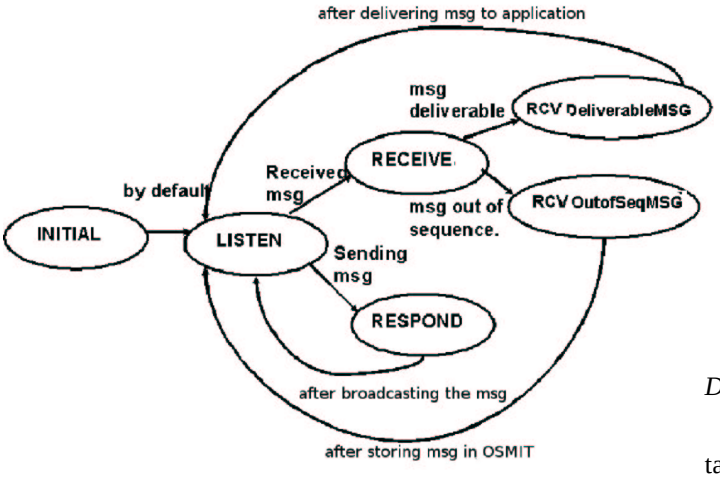

Fig. 8. State Diagram of the protocol

1) **INITIAL STATE:**
   At every process, set *seqno* to zero, set root of OT to *OTR* and go to LISTEN STATE.
2) **LISTEN STATE:**
   Listen until a message is received or application wants to respond to a message.
   if message is received **then**
   go to RECEIVE state
   else if application sends a message to the group **then**
   go to RESPOND state.
   end if
3) **RECEIVE STATE:** Process i on receiving a message $M =< mid_c, mid_p, gid, data >$,
   if mid_p = OTR or $mid_p \in OT$ **then**
   go to RCVDeliverableMSG STATE.
   else
   go to RCVOutSequenceMSG STATE
   end if
4) **RCVDeliverableMSG STATE:**
   a) Call the UpdateOT operation described below with received message M as its parameter.
   b) UpdateOT(M)
      i) Append $M \cdot mid_c$ into OT_i as a child node of $M \cdot mid_p$.
      ii) Deliver the $M \cdot data$ to the application.
      iii) /* Let Msg represents a message in $OSMS_i$ and $Msg \cdot mid_p$, $Msg \cdot mid_c$ represent the mid_p, mid_c values of the message Msg respectively. */
      for each message $Msg \in OSMS_i$ having $Msg \cdot mid_p == M \cdot mid_c$ **do**
      UpdateOT(Msg)
      Remove message Msg from $OSMS_i$
      end for
   c) go to LISTEN state.
5) **RCVOutSequenceMSG STATE:**
   Insert $< mid_c, mid_p, data >$ in $OSMS_i$ and go to LISTEN state.
6) **RESPOND STATE:** When application at process i responds to a message with identity mid_p, then,
   a) $seqno_i = seqno_i + 1$
   b) $mid_i =< pid_i, seqno_i >$
   c) $broadcasts :< mid_i, mid_p, data >$
   d) go to LISTEN state.

   If message is not related to prior messages then mid_p is OTR.

D. Java Implementation

The class diagram of the protocol and the java implementation of data structures are described Appendix.

V. Group Join and Leave Protocols

In this section we present GJLP protocol, for processes that have newly entered the network, to become aware of the various group applications currently operating in the broadcast network and to join in any of these groups. The GJLP protocol is also for processes, that rejoined the network after leaving it temporarily, to know the list of new group applications currently operating in the network and also to update the membership list of the groups that it is already member of.

- **At Sender:** The process sends advertisement message to inform its presence to the processes present in the network.
- **At every process on receiving advertisement :** Every process, including the process that sent the advertisement, on receiving advertisement message sends information about the groups, that they are member of, in the form of a list of group identities and their members identities.
- **At every process on receiving information about groups:** Every process on receiving them, if finds any new group that they are not aware of, then they send it to the application. If application wants to join a group, the process broadcasts its identity and the identity of the group that it is joining. Every member of the group on receiving it updates the members list of the group.
- **For a process to leave a group:** The process, for leaving a group, broadcasts its identity and the identity of the group that it is leaving. Every member of the group on receiving it updates the members list of the group.

A. Notations, Message Format and Data Structures

1) Notations:

- $< cMemList >$: denotes list of identities of the processes that are current members of a group.
- $< lMemList >$: denotes list of identities of the processes that have left the group.
- $< grpInfo >$: denotes the information of a group. It is of the format $<$ *gid, desc,* $cMemList, Memlist >$ and *desc* denotes the description about the group.

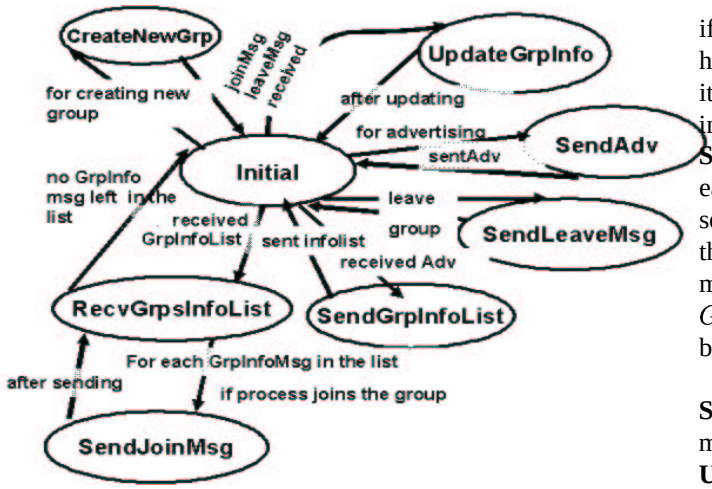

Fig. 9. Group Join/Leave Protocol state diagram

*2) **Messages Format**:*

- $< advMsg >$: denotes advertisement message sent by process. It contains the identity of the process in the format $< pid >$
- $< grpsInfoList >$: denotes information about the various groups. It is of the format: $< list\,of < grpInfo >>$.
- $< joinMsg >$: It is of the format $< pid, gid >$ and denotes message sent by process whose identity is pid to inform the members of the group gid that it is joining the group.
- $< leaveMsg >$: It is of the format $< pid, gid >$ and denotes message sent by process pid to inform the members of the group gid that it is leaving the group.

*3) **Data Structures**:* Every process maintains the following data structure:

- $GrpsInfoIndexTable_i$ $(GIIT_i)$.Each row of the table $GIIT_i$ at process **i** contains $grpInfo$ of a group and the row is indexed by the identity of the corresponding group $grpInfo \cdot gid$ (i.e. row $GIIT_i[grpInfo \cdot gid]$ contains $grpInfo$). If the process i is not member of group whose identity is gid then the row of $GIIT_i$ indexed by gid contains **N/A** value. (i.e. if **i** is not member of group gid then $GIIT_i[gid]$ is **N/A**)

B. Protocol Actions

The state diagram of the protocol at process **i** is as shown in the Fig 9. Brief explanation of protocol action is as follows: The process for advertising its presence to the group goes from the **Intial** state to **SendAdv** state. In this state, it creates advertisement message $adMsg$ and broadcasts it. After broadcasting, it returns to the **Initial** state. Every process including the process that sent the advertisement on receiving advertisement goes to **SendGrpInfoList** state, creates $grpsInfoList$ from the $grpInfo$ present in the rows of their **GIIT** and broadcasts it.

Every process present in the broadcast network on receiving $grpsInfoList$, goes to **RecvGrpsInfoList** state. In this state, if process **i** finds any $grpInfo$ in $grpsInfoList$ that do not have an entry in its $GIIT_i$, then it delivers $grpInfo \cdot desc$ to its application. If the application at this process is interested in joining any of these groups then the process goes to **SendJoinMsg** state. In this state it creates a $joinMsg$ for each group and broadcasts them. On receiving $joinMsg$ sent by the process for joining a group, every member of the group goes to **UpdateGrpInfo** state. In this state every member of the group gets $grpInfo$ of the group from its $GIIT[joinMsg \cdot gid]$ (by indexing operation) and updates it by appending the $joinMsg \cdot pid$ to the $grpInfo \cdot cMemList$.

If process **i** is leaving a group, the process goes to **SendLeaveMsg** state and broadcasts $< leaveMsg >$. Each member of the group on receiving $< leaveMsg >$ goes to **UpdateGrpInfo** state. In this state a process **i** gets $grpInfo$ from its $GIIT_i[leaveMsg \cdot gid]$ and updates $grpInfo$ by transferring $leaveMsg \cdot pid$ from $grpInfo \cdot cMemList$ to $grpInfo \cdot lMemList$.

A more detailed description of the protocol actions in each state at process i is as follows:

*1) **Initial STATE**:*

if process newly enters network or rejoined network after leaving it temporarily **then**
  go to **SendAdv** state for sending its advertisement.
else if process receives $< advMsg >$ sent by any process **then**
  go to **SendGrpsInfoList** state.
else if process receives $grpsInfoList$ from any process **then**
  go to **RecvGrpsInfoList** state.
else if process receives $joinMsg$ or $leaveMsg$ **then**
  go to **UpdateGrpInfo** state.
else if process wants to leave a group **then**
  go to **SendLeaveMsg** state.
else if process wants to create a new group **then**
  go to **CreateNewGrp** state.
end if

*2) **SendAdv STATE**:*

- Delete row of $GIIT_i$ containing the entry *N/A*
- For process i to advertise its presence to the group, creates $< advMsg_i >$ = $< pid >$, broadcasts it and goes back to **Intial** state.

*3) **SendGrpInfoList STATE**:* Process **i** and every other processes including the process that sent $< advMsg >$ on receiving $< advMsg >$ does the following,

/*Create $grpsInfoList$ by doing the following.*/
for each $grpInfo$ present in $GIIT_i$ **do**
  add $grpInfo$ to $< grpsInfoList >$.
end for
Broadcast $grpsInfoList$

*4) **RecvGrpsInfoList STATE**:* Process **i** on receiving $grpsInfoList$:

for each $grpInfo$ present in $grpsInfoList$ **do**
  if there is an entry in $GIIT_i$ indexed by $grpInfo \cdot gid$ **then**

if entry is **N/A** **then**
Discard the $grpInfo$
else if entry is group information $gInfo$ **then**
Update the $gInfo$ by doing the following.
Insert every pid present in $grpInfo \cdot lMemList$ in $gInfo \cdot lMemList$ if pid does not exist in it and remove pid from $gInfo \cdot cMemList$ if pid exists. Similarly insert every pid present in $grpInfo \cdot cMemList$ in $gInfo \cdot cMemList$ if it does not exist in $gInfo \cdot cMemList$ and $gInfo \cdot lMemList$.
end if
else
Deliver $grpInfo \cdot desc$ of the group to the application.
if application wants to join the group **then**
Process **i** creates a new row in $GIIT_i$, inserts in it $grpInfo$ and sets index key value to $grpInfo \cdot gid$.
go to **SendJoinMsg** state.
else
Discard the $grpInfo$
end if
end if
end for

5) ***SendJoinMsg STATE****:* To join group gid process i creates $joinMsg$ with $< pid, gid >$ and broadcast it.

6) ***UpdateGrpInfo STATE****:* Process i on receiving $joinMsg$ or $leaveMsg$,

if received message is $joinMsg$ **then**
if process i is member of group $joinMsg \cdot gid$ **then**
Get the $grpInfo$ from $GIIT_i[joinMsg \cdot gid]$ and add the $joinMsg \cdot pid$ to $grpInfo \cdot cMemInfo$
end if
else if received message is $leaveMsg$ **then**
if process i is member of group $leaveMsg \cdot gid$ **then**
Get the $grpInfo$ from $GIIT_i[joinMsg \cdot gid]$ and remove the $leaveMsg \cdot pid$ from $grpInfo \cdot cMemList$ and add it $grpInfo \cdot lMemList$.
end if
end if

7) ***SendLeaveMsg STATE****:* For process to leave group whose identity is *gid*, it destroys MOP, MSP instances of the group, creates *leaveMsg* in the format <*pid*,*gid*> and broadcasts it.

8) ***CreateNewGrp STATE****:* The process creates a unique identity gid for the new group, creates $grpInfo$ and sets $grpInfo \cdot gid$ to gid. It appends its identity pid to $grpInfo \cdot cMemList$ and broadcasts $grpInfoList$ containing $grpInfo$.

C. Java Implementation Details

We have implemented the protocol in Java. The class diagram of the protocol and its methods are described in Appendix.

VI. Member Synchronization Protocol (MSP)

A process that newly joins a group or process that rejoins the group after leaving the group temporarily executes MSP to recover the messages that were sent to the group during its absence. We assume that every member of the group logs the messages that it has received from the group. MSP protocol is as follows:

- **At the sender:** The process sends synchronization request message *SyncReqMsg* that contains the list of the messages that it has so far received from the group.
- **At the receivers:** On receiving a *SyncReqMsg* from a process, every process (excluding the process that sent *SyncReqMsg*) in the group creates a repository of message identities. The repository contains the identities of messages that it has received from the group excluding the messages present in *SyncReqMsg*. It starts a counter with a random value. When the counter expires, it broadcasts a synchronization response message *SyncRespMsg*, containing the messages whose identities are present in repository, to the group and deletes the repository. While counting, for every *SyncRespMsg* that it receives from any member of the group, it deletes the identities of those messages from repository that are present in received *SyncRespMsg*. The process that sent *SyncReqMsg* on receiving *SyncRespMsg*, sends the messages present in message list of *SyncRespMsg* to the GroupManager which will be subsequently delivered to application.

A. Notations, Message Format and Data Structures

1) ***Notations****:*

- $< SyncSeqno >$: denotes a sequence counter. It is initialized to zero and incremented before *SyncReqMsg*.
- $< SyncMsgId >$: denotes the identity of *SyncReqMsg* or *SyncRespMsg* and it is in the format $< pid, syncSeqno, gid >$
- $< SyncMsgList >$: denotes list of messages. Each message $< Msg >$ is in the format $< mid_c, mid_p, data >$ where mid_c, mid_p, $data$ are identity of message, identity of its parent message and application $data$ as described in Section 4.

2) ***Message Format****:*

- $< SyncReqMsg_i >$ denotes *SyncReqMsg* sent by process i and it is in the format $< SyncMsgId_i, SyncMsgList >$.
- $< SyncRespMsg_{ij} >$ denotes *SyncRespMsg* sent by process j in response to $< SyncReqMsg_i >$ and it is in the format $< pid_j, SyncMsgId_i, SyncMsgList >$.

3) ***Data Structure****:* Every process maintains following data structures:

- **Group Messages List** $(GrpMsgList_{gid})$ The data structure stores every message that is sent to group with *gid* in the list so that when a new member requires synchronization, messages present in this list will be transferred to it. Each message stored in the list is of format $< mid_c, mid_p, data >$. If the process has memory constraints and if the application $data$ is not very significant than it stores only mid_c, mid_p because these identities are used by MOP for ordering purpose.

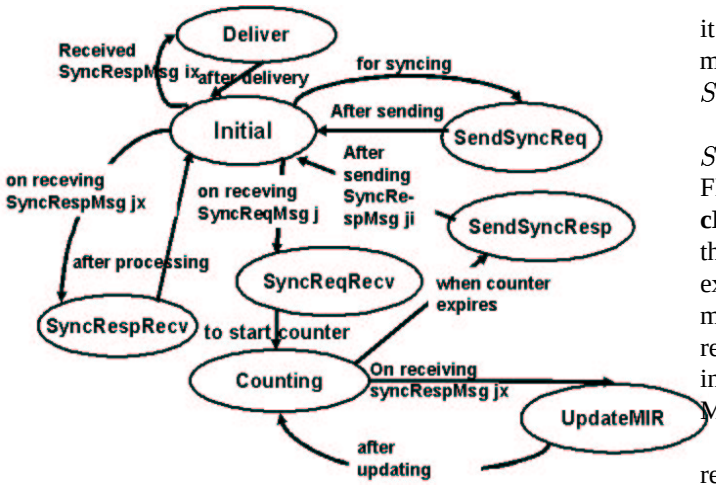

Fig. 10. Member Synchronization Protocol

- **Message Identities Repository (MIR)**: The data structure MIR stores identities of messages. It is created for temporary period on receiving $SyncReqMsg$ and deleted after the process responds to it by sending $SyncRespMsg$.
- **Process Sync Status Index Table ($PSSIT_i$)**: The rows of index table $PSSIT_i$ at process **i** stores for every process the latest *SyncSeqno* of *SyncReqMsg* among the *SyncReqMsg* messages that it has so far received from the process. Each entry $SyncSeqno$ of $PSSIT_i$ is indexed by *pid* of the process.
- **MIRIndexTable**: The index table entry points to MIR indexed by $SyncMsgId$.

B. State Diagram

The state diagram of the protocol at process **i** is as shown in fig10.

The process **i** in the **Initial** state goes to **SendSyncReq** state and sends $SyncReqMsg_i$. If process **i** receives $SyncReqMsg_j$ from process **j** it goes to **SyncReqRecv** state. Checks if the received $SyncReqMsg_j$ is latest request message from process **j**(because network layer may not follow FIFO order in message delivery) by comparing the $SyncReqMsg_j \cdot SyncSeqno$ with $SyncSeqno$ present in $PSSIT_i[SyncReqMsg_j \cdot SyncMsgId \cdot pid]$ and updates this entry in $PSSIT_i$ with $SyncReqMsg_j \cdot SyncSeqno$ if the request message is latest message. If the received message is latest message, it creates MIR (if it does not exist in $MIRIndexTable$ when indexed by key $SyncReqMsg_j.SyncMsgId$), stores in MIR the identities of those messages present in $(GrpMsgList_{gid})$ (where gid is $SyncMsgId \cdot gid$) and not in $SyncReqMsg_j \cdot SyncMsgList$ and goes to **Counting** state. In this state it starts a counter with random initial value and keeps decrementing. If the counting reaches zero it goes to **SendSyncResp** state and creates $SyncRespMsg_{ji}$ containing the list of messages whose identities are present in MIR. It broadcasts $SyncRespMsg_{ji}$ and deletes MIR. While counting if it receives $SyncRespMsg_{jx}$ it goes to **UpdateMIR** state deletes the identities of those messages from MIR that are present in $SyncRespMsg_{jx} \cdot SyncMsgList$.

If process **i** receives the $SyncRespMsg_{jx}$ before $SyncReqMsg_j$ because the network layer may not follow FIFO order in delivering messages then it goes to **SyncRespRecv** state. If the received message is latest message then it does the following. Creates MIR if one does not exist in MIRIndexTable, stores in MIR the identities of those messages that are present in $GrpMsgList_{gid}$ and not in received $SyncRespMsg_{jx}$ message. If MIR already exists in MIRIndexTable then it deletes identities of messages from MIR that are present in $SyncRespMsg_{jx} \cdot SyncMsgList$.

If process **i** receives $SyncRespMsg_{ix}$ in response to its request message $SyncReqMsg_i$ it goes to **Deliver** state and delivers the messages present in $SyncRespMsg_{ix} \cdot SyncMsgList$ to GroupManager.

A more detailed description of the protocol at process i is given below:

*1) **Initial STATE**:*

Contents of $PSSIT_i$ are made empty and SyncSeqno is set to zero.
if process wants to sync with rest of group members of group gid **then**
  Go to **SendSyncReq** state
else if a $SyncReqMsg_j$ is received **then**
  Go to **SyncReqRecv** state
else if $SyncRespMsg_{ik}$ is received **then**
  Go to **DeliverMsg** state
else if $SyncRespMsg_{jk}$ is received **then**
  Go to **SyncResRecv** state
end if

*2) **SendSyncReq STATE**:*

- Increments SyncSeqno.
- Creates a $< SyncMsgId_i >$ with $< pid >$, $< gid >$ and $< SyncSeqno >$.
- Creates $< SyncMsgList >$ with the list of the messages present in $GrpMsgList_{gid}$.
- Creates $< SyncReqMsg_i >$ in format $< SyncMsgId_i, SyncMsgList >$ and broadcast it.

*3) **SyncReqRecv STATE**:*

On receiving $SyncReqMsg_j$, let $msgId$, $msgList$, gid represent $SyncReqMsg_j \cdot SyncMsgId$, $SyncReqMsg_j \cdot SyncMsgList$ and $SyncReqMsg_j \cdot SyncMsgId \cdot gid$ respectively.
if $PSSIT_i[msgId \cdot pid] < msgId \cdot SyncSeqno$ **then**
  Create MIR with identities of messages present in $GrpMsgList_{gid}$ and not in $msgList$ and set $MIRIndexTable[msgId]$ to MIR
  Set $PSSIT_i[msgId \cdot pid]$ to $msgId \cdot SyncSeqno$
  Go to **Counting** state
else if $PSSIT_i[msgId \cdot pid] = msgId \cdot SyncSeqno$ **then**
  Get MIR from MIRIndexTable[$msgId$]
  Delete message identities from MIR whose messages present in $msgList$

Go to **Counting** state
else
Discard $SyncReqMsg_{jk}$
end if

*4) **Counting STATE**:*

Choose a random counter value.
while counter does not reach zero **do**
Decrement the counter by one.
if the process receives $SyncMsgRes_{jx}$ from some process x **then**
Get MIR from $MIRIndexTable[SyncMsgRes_{jx} \cdot SyncMsgId]$
go to **UpdateMIR** State
end if
end while
if counter reached zero **then**
Go to **SendSyncResp** state
end if

*5) **UpdateMIR STATE**:*

- Deletes the identities from MIR whose messages are present in $SyncRespMsg_{jx} \cdot SyncMsgList$.
- Go to *while loop* at statement 2 of **Counting** state.

*6) **SendSyncResp STATE**:*

- Create $SyncMsgList$ with the list of messages from $GrpMsgList_{gid}$ whose identities are present in MIR.
- Create $SyncRespMsg_{ji}$ in the format $< SyncMsgId, SyncMsgList >$.
- Delete MIR and its reference from MIRIndexTable.
- Broadcast the $SyncRespMsg_{ji}$.

*7) **Deliver STATE**:* On receiving $SyncRespMsg_{ij}$,deliver the messages present in $SyncMsgList$ of the received $SyncRespMsg_{ij}$ to the GroupManager which will subsequently deliver them to the application.

*8) **SyncRespRecv STATE**:*

On receiving $SyncRespMsg_{jx}$, let $msgId$, $msgList$, gid represent $SyncRespMsg_{jx} \cdot SyncMsgId$, $SyncRespMsg_{jx} \cdot SyncMsgList$ and $SyncRespMsg_{jx} \cdot SyncMsgId \cdot gid$ respectively.

if $PSSIT[msgId \cdot pid] < msgId \cdot SyncSeqno$ **then**
Create MIR with identities of messages present in $GrpMsgList_{gid}$ and not in $msgList$ and set MIRIndexTable[msgId] to MIR
Set $PSSIT[msgId \cdot pid]$ to $msgId \cdot SyncSeqno$
else if $PSSIT[msgId \cdot pid] = msgId \cdot SyncSeqno$ **then**
Get MIR from MIRIndexTable[msgId] if exists
Delete message identities from MIR whose messages present in $msgList$
else
Discard $SyncRespMsg_{jx}$
end if

C. Java Implementation

The class diagram of the protocol and the java implementation of data structures are described Appendix.

VII. Related Work: Anhinga Project

The Anhinga Project [1] is an infrastructure for building distributed applications involving many-to-many communication in an ad hoc network of proximal mobile wireless devices. Its architecture consists of Many2Many Invocation (M2MI) mechanism and Many2Many Protocol(M2MP).

*1) **M2MI**:* Remote method Invocation(RMI) can be viewed as an object oriented abstraction of point-to-point communication: what looks like a method call in fact a message sent and a response sent back. In contrast to RMI ,M2MI provides an object oriented method call abstraction based on broadcasting. An M2MI-based application broadcasts method invocation, which are received and performed by many objects in many target devices simultaneously. The paper [1] describes it in detail.

*2) **M2MP**:* The M2MP is similar to our GJLP protocol. If more than one group application exists then M2MP lets application join a group and leave group. It also regulates the received packets from the broadcast layer to appropriate group application.

Although Anhinga project provides Object oriented method call abstraction for developing many to many communication applications, it does not provide support for message ordering and recovery of lost messages.

VIII. Conclusion

We have presented M2MC, a new distributive computing middleware designed to support collaborative applications running on devices connected by broadcast networks. M2MC is useful for building a broad range of multi-user applications like multiplayer games, conversations, group ware systems. M2MC does not rely on central servers and its component protocols MOP,MSP,GJLP act together for communicating in a distributed manner. We have described the specification details and Java implementation details of these protocols. We have discussed Threaded Chat Application developed using M2MC.

References

[1] H.Bischof. A.Kaminsky. Many-to-many invocation: A new framework for building collaborative applications in ad hoc networks. *CSCW 2002 Workshop on Ad Hoc Communication and Collaboration in Ubiquitous Computing Environments, New Orleans, Louisiana, USA,*, 2002.

[2] D.R.Cheriton and D.Skeen. Understanding the limitations of causally and totally ordered communication. *In the Preceedings of 14th ACM Symposium on Operating System Principles*, pages 44–57, 1993.

[3] R.Vitenberg. G.Chockler, I.Keidar. Group communication specifications:a comprehensive study. *ACM Computing Surv. 9,2(Feb.), 427-469*, 2001.

[4] F. Viegas J. Donath, K. Karahalios. Visualizing conversation. *Proceedings of HICSS-32*, 1999.

[5] L.Lamport. Using time instead of time-outs in fault-tolerant systems. *ACM Trans. Program. Lang. Syst. 6,256-280*, 1984.

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

[7] B.Burkhalter M.Smith, JJ. Cadiz. Conversion trees and threaded chats. *ACM Magazine*, 2000.

[8] P.Urban X.Defago, A.Schiper. Comparitive performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. Inf. Syst. E86-D,12*, 2003.

[9] P.Urban. X.Defago, A.Schiper. Total order broadcast and multicast algorithms: Taxonomy and survery. *ACM Computing Surveys,Vol. 36,No. 4 pp.372-421*, December 2004.

IX. Appendix: Java Implementation of M2MC middleware Layer

A. System Environment

We have implemented the middleware protocols in Java language. We have used the *Multicast* socket provided by the Java.io package for multicasting over IP address. We discuss the implementation details of each protocol using their class diagrams.

B. Message Ordering Protocol

We have implemented the protocol as Java APIs. The class diagram of the protocol is as shown in Fig.11.

*1) class**OutSeqMsgList**:* implements the data structure (OSMS) in the form of linked list called *OsmMsgList* with each node of the list pointing to object of *Msg* that stores mid_p, mid_c, $data$ of application message. The *insertInOsml(midc,midp,data)* creates object of *Msg* and inserts in *OsmMsgList*. The *getInSeqMsgList(midc)* retrieves all the messages (Msg) from *osmMsgList* for which *midc* is semantically before them either directly or transitively and returns them to the called function in the form of linked list of messages in S_b order.

*2) class**OrderingList**:* implements the data structure OrderingTree as collection of linked lists called *seqnoList*. One *seqnoList* is maintained for every member of the group to store the sequence numbers of the messages sent by the member. If sequence numbers are continous only the first and last value are stored to save space. A java.util. HashMap called *pidMap* maps the identity of a process (pid) to the linked list seqnoList corresponding to the process. The *isPresent(midp)* returns true if *midp.seqno* is present in *seqnoList* of process *midp.pid* and the *insert(midc)* inserts *midc.seqno* in *seqnoList* of process *midc.pid*.

*3) class **MessageOrderingProtocol**:* is called by GroupManger before sending message to the group. It is also called after receiving a message from the group for ordering message in S_b order before sending it to the application. The *send(midp, data)* creates identity *midc* for application data and sends it to the GroupManager in the form of DataPacket object containing attributes *midc*, *midp*, *gid*, *data*. The *receive(DataPacket dpkt)* calls *isPresent(dpkt.midp)* of *OrderingList* class, if it receives true then calls *getInSeqMsgList(midc)* and gets the linked list of messages. It calls *insert(mid)* of OrderingList class and inserts the identities of these messages in OrderingList. It delivers these messages to the GroupManager in S_b order. If *isPresent(dpkt.midp)* returns false then calls *insertInOsml(dpkt.midc,dpkt.midp,dpkt.data)* to insert the out of sequence message in OSMS.

C. Group Join/Leave Protocol

The class diagram of the protocol in the Fig12 shows salient attributes and methods of the class GrpJLProtocol.

class: GrpJLProtocol

The class implements the data structure *GIIT* as object of java.util.HashMap called *GrpInfoMap*. The key of the *GrpInfoMap* is the object of group identity class *Gid* containing unique identity for each group and entry of the map is object of *GrpInfo* containing information about the group in its attributes *gid, desc* and java.util.LinkedList objects *cMemList, lMemList*.

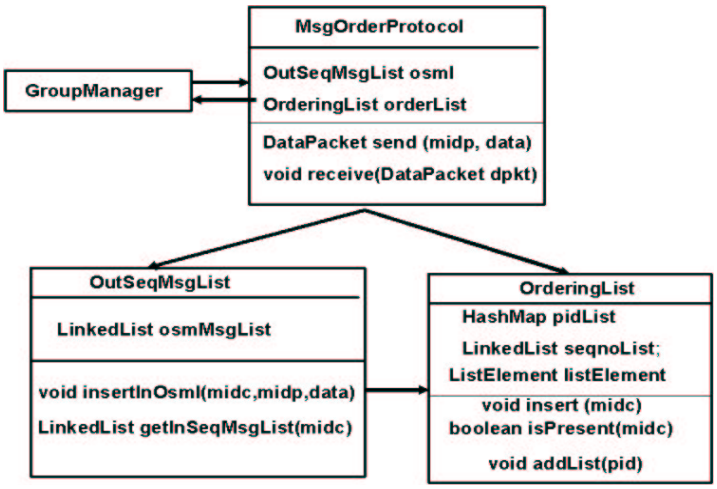

Fig. 11. Message Ordering Protocol class diagram

*1) **sendAdMsg()**:* deletes all entries containing value *N/A* in grpMsgList and sends an object of *AdMsg* class containing attribute *pid*.

*2) **LinkedList getGrpInfoList()**:* returns linked list called *GrpsInfoList* whose nodes contain objects of *GrpInfo* which are present in *GrpInfoMap*.

*3) **adMsgReceived(AdMsg adMsg)**:* at every process (including the process that sent the message) on receiving adMsg calls *getGrpInfoList()* , gets the linked list of the *GrpInfoList* and broadcasts it.

*4) **updateGrpInfoMap(GrpInfo recvGrpInfo)**:* gets the *grpInfo* from *GrpInfoMap* by indexing with key value *recvGrpInfo.gid*. The method updates *grpInfo.lMemList* and *grpInfo.cMemList* by doing the following. The process inserts each *pid* present in *recvGrpInfo.lMemList*, in *grpInfo.lMemList*. Also if *pid* is present in *grpInfo.cMemList* then process removes *pid* from *grpInfo.cMemList*. The process inserts each *pid* present in *recvGrpInfo.cMemList*, in *grpInfo.cMemList* if it does not exist either in *grpInfo.cMemList* or in *grpInfo.lMemList*.

*5) **void grpInfoListRecv(grpInfoList)**:* For each *grpInfo* object present in *grpInfoList*, it does the following. If there is an entry in *GrpInfoMap* with key value *grpInfo.gid*, then it calls *updateGrpInfoMap(grpInfo)*. Otherwise it informs the user about the new group by delivering the *grpInfo* to the application through GroupManager. If application wants to join the group then calls *joinThisGroup(grpInfo)* of GroupManager which calls *joinGroup(grpInfo)* else it calls *rejectGrp(grpInfo)*

*6) **void joinGrp(grpInfo)**:* updates *GrpInfoMap* by adding the entry *grpInfo*, its key *grpInfo.gid* and calls *sendJoinMsg(grpInfo.gid)*.

*7) **void rejectGrp(grpInfo)**:* updates *GrpInfoMap* by adding the entry *N/A*, its key *grpInfo.gid*.

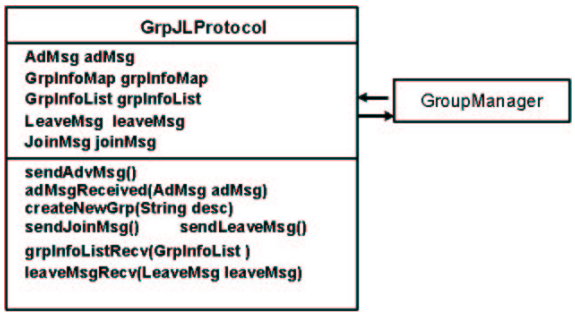

Fig. 12. Group Join/Leave Protocol

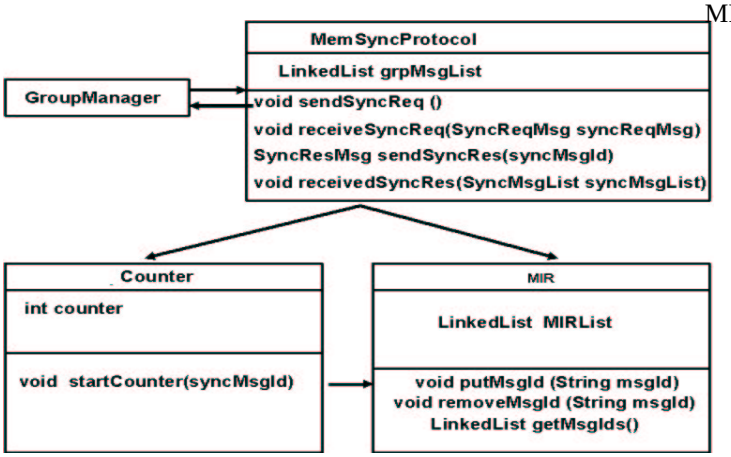

Fig. 13. Member Synchronization Protocol

8) ***sendJoinMsg()****:* creates an object of *JoinMsg* class containing attributes *pid, gid* and broadcasts it.

9) ***sendLeaveMsg(gid)****:* creates an object of *LeaveMsg* class with fields *pid, gid* and broadcasts it. It also calls *GroupManager* for destroying the objects of MSP, MOP of the group.

10) ***joinMsgReceived(joinMsg)****:* gets the *grpInfo* from *GrpInfoMap* by indexing with *joinMsg.gid* if exists, and inserts joinMsg.pid in *grpInfo.cMemList*.

11) ***leaveMsgReceived(leaveMsg)****:* gets the *grpInfo* object from *GrpInfoMap* by indexing with leaveMsg.gid if exists and inserts *leaveMsg.pid* in *grpInfo.lMemList*. It also deletes *leaveMsg.pid* from *grpInfo.cMemList*

12) ***createNewGroup(desc)****:* creates new group by doing the following. It creates unique identity *gid* for the group by incrementing *grpSeqno* and appending process identity *pid* to it. It creates object *grpInfo* of *GrpInfo* class sets *grpInfo.gid*, *grpInfo.desc* to *gid*, *desc* respectively. It inserts process identity *pid* in *grpInfo.cMemList* linked list and updates *GrpInfoMap* by adding the entry *grpInfo* and its key *gid*. It creates *grpInfoList* containing *grpInfo* and broadcasts it.

D. Member Synchronization Protocol

The class diagram of the protocol in Fig13 shows the salient methods and attributes of each class.

1) ***class:MemberSyncProtocol****:* The MemberSyncProtocol implements the data structure **GrpMsgList** as a java.util.LinkedList object called grpMsgList with each node of the linked list pointing to object of Msg. (Msg has fields midc,midp,data). HashMap (**ProcessSyncStatusMap(PSSM)**) implements the data structure **ProcessSyncStatusIndexTable(PSSIT)** with each entry of the map is indexed by key value **pid** of process and each entry contains the **SyncSeqno** of latest **SyncReqMsg** sent by process with identity pid. MIR is implemented as LinkedList called **MIRList**. Another HashMap **MIRMap** implements **MIRIndexTable** for storing the objects of MIRList indexed by key value $SyncMsgId$.

- **sendSyncReq()** increments its syncSeqno and creates object of SyncMsgId class with fields pid, syncSeqno, gid. It creates object of SyncMsgList containing the linked list of messages that are present in **grpMsgList** and broadcasts the object of SyncReqMsg containing objects of SyncMsgId and SyncMsgList.
- **updateGrpMsgList(SyncMsgList syncMsgList)** updates the grpMsgList by appending messages to grpMsgList that are present in syncMsgList and not in grpMsgList. These messages are subsequently delivered to application.
- **receiveSyncReq(syncReqMsg)** at every process (except process that sent syncReqMsg) calls updateGrpMsgList(syncReqMsg.syncMsgList). It gets the syncSeqno of latest syncReqMsg sent by process syncReqMsg.syncMsgId.pid from PSSM and if syncReqMsg.syncMsgId.syncSeqno is less than syncSeqno then it discards syncReqMsg, if greater than syncSeqno then creates object of MIRList (with identities of messages present in **grpMsgList** and not in syncReqMsg.syncMsgList). If syncSeqno is equal to syncReqMsg.syncMsgId.syncSeqno then get the object of MIRList from MIRMap by indexing with key value syncReqMsg.syncMsgId and update it by removing the identities of those messages from MIRList that are present in syncReqMsg.syncMsgList. It creates object of counter class and starts the counter thread.
- **receiveSyncResp(syncRespMsg)** at process **i** checks if syncRespMsg.syncMsgId.pid is pid_i and calls updateGrpMsgList(syncResMsg.syncMsgList). Otherwise gets syncSeqno from PSSM and if syncRespmsg.syncMsgId.syncSeqno is less than syncSeqno then it discards syncReqMsg, if greater than syncSeqno then creates object of MIRList (with identities of messages present in **grpMsgList** and not in syncReqMsg.syncMsgList). If syncSeqno is equal to syncReqMsg.syncMsgId.syncSeqno then get the object of MIRList if one exists from MIRMap by indexing with key value syncReqMsg.syncMsgId and update it by removing the identities of those messages from MIRList that are present in syncReqMsg.syncMsgList.
- **sendSyncResp(syncMsgId)** gets the MIRList object

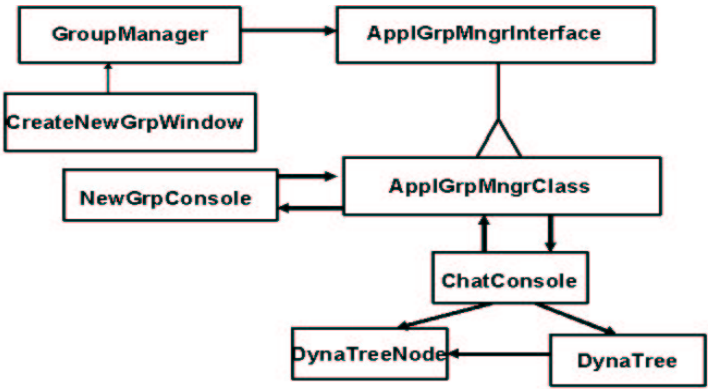

Fig. 14. Threaded Chat Application Architecture

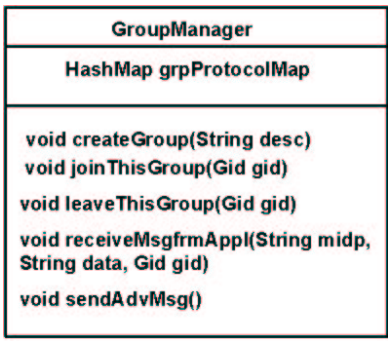

Fig. 15. Group Manager class

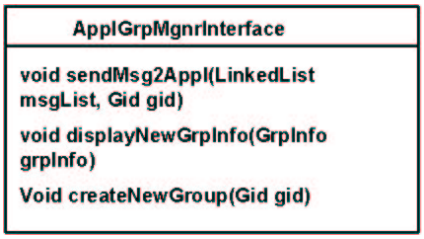

Fig. 16. Interface for application developer

from MIRMap by indexing with key syncMsgId and creates syncMsgList with the list of messages present in MIRList. It creates syncRespMsg with syncMsgList, syncMsgId and broadcasts syncRespMsg to the group. It destroys MIRList and updates MIRMap.

2) ***class:Counter****:* inherits Thread class. Its only method is startCounter(syncMsgId). The method takes random value and keeps decrementing. If reaches counter value reaches zero it calls sendSyncResp(syncMsgId).

3) ***class: MIR*** *:* maintains LinkedList MIRList. It contains methods like putMsgId(mid), removeMsgId(mid), getMsgIds() for putting, removing, sending identities respectively.

E. ***Application development using M2MC APIs***

For developing application, the application developer uses the APIs provided by GroupManager class and implements the methods of interface ApplGrpManagerInterface.

The class **GroupManager** (Fig.15) of the M2MC provides the following method interfaces for the application developers.It maintains object of java.util.HashMap called *grpProtocolMap* for mapping identity of a group *gid* to its instances of classes MsgOrderProtocol and MemSyncProtocol.

1) void ***CreateGroup(String desc)****:* is for the application developer for creating a new group. The argument **desc** is description about the group. It gets identity for the new group from object of GrpJLProtocol method, creates instances of classes MsgOrderProtocol and MemSyncProtocol, and updates *grpProtocolMap* to map the identity of the group to the instances of classes. It calls *getGrpInfoList()* of *GrpJLProtocol* class, gets *grpsInfoList* that includes *grpInfo* of new group and broadcasts it.

2) void ***joinThisGroup(grpInfo)****:* is called for joining an existing group whose identity is grpInfo.gid. It creates instances of MsgOrderProtocol, MemSyncProtocol, updates *grpProtocolMap* and calls *joinGrp(grpInfo)* of GrpJLProtocol class.

3) void ***leaveThisGroup(gid)****:* is called if the process wants to leave from group whose identity is gid. It calls *sendLeaveMsg(gid)* method of *GrpJLProtocol*.

4) void ***receiveMsgfrmAppl(midp, data, gid)****:* is called when the user at group application wants to broadcast application $data$ to the group in response to a message with identity mid_p. It calls *send(midp,data)* object of class MsgOrderProtocol corresponding to group with identity *gid*.

5) void ***sendAdvMsg()****:* for advertising the presence of process in the network. The method calls *sendAdMsg()* of GrpJLProtocol.

The interface ApplGrpMgnrInterface provides the following APIs. These APIs are called by the methods of GroupManager and other classes of M2MC. The application developer implements the methods provided in the interface based on application logic.

6) void ***sendMsg2Appl(LinkedList msgList, String gid)****:* The method is called by GroupManager methods for giving the application the list of messages that the process has received from the group of group identity gid.

7) void ***displayNewGroupInfo(GrpInfo grpInfo)****:* is called by GroupManager, for sending identity and description of new group to application.

8) void ***createdNewGroup(String gid)****:* is called by GroupManager, for sending the identity of newly created group. The application developer can write the application specific code (that has to be performed when a new application group has been created) by implementing this method.

The class diagram of the Threaded Chat application is shown in the Fig 14. The classes ApplGrpMngrClass, GroupInfoWindow, ChatConsole, DynaTreeNode, DynaTree, represent the application logic. The classes ChatConsole, DynaTreeNode, DynaTree, GroupInfoWindow provide GUI. The discussion of these classes is beyond the scope of the paper.