M2MC: Middleware for Many to Many Communication over broadcast networks

Dissertation

submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Chaitanya Krishna Bhavanasi

(Roll no. 03329003)

under the guidance of

Prof. Sridhar Iyer

Kanwal Rekhi School of Information Technology

Indian Institute of Technology Bombay

2005

Dedicated to Prof.Sridhar Iyer for his excellent guidance.

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.

M2MC architecture consists of Messages Ordering protocol, Member Synchronization protocol, and protocols for processes to join and leave the groups. We emphasized on Message Ordering protocol as it is a key component in developing group communication applications. Hence we proposed a new message ordering called S_b ordering that orders the messages based on their semantic relationship as specified by the users.

Some salient features of M2MC are: Unlike existing middleware architectures that rely on central servers, the M2MC is truly distributed protocol and hence application developed using M2MC does not require central servers. Being broadcast oriented, M2MC does not require any resource consuming routing protocols. Distributed applications development is simplified by M2MC APIs.

Contents

List of Figures

Chapter 1

Introduction

1.1 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 of network layer 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 Fig1.1a, 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 Fig1.1b, 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 Fig1.1c 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 many to many communication is a form of group communication paradigm with devices present in the network forming multiple overlapping groups as shown in Fig1.1d. The middleware architecture proposed in this thesis is especially to support applications with many−to−many communication patterns, although it also supports other communication patterns.

The thesis describes a set of protocols for many 2 many communication paradigm, for building collaborative applications like multiplayer games, chat applications etc. that run in wireless proximal ad hoc networks of fixed or mobile devices or wired or wireless devices connected by hybrid network. The Fig.1.2 shows the relative position of middleware protocols with respect to application and the network protocols. The middleware consists of Messages Ordering protocol, Messages Synchronization protocol, and protocols for

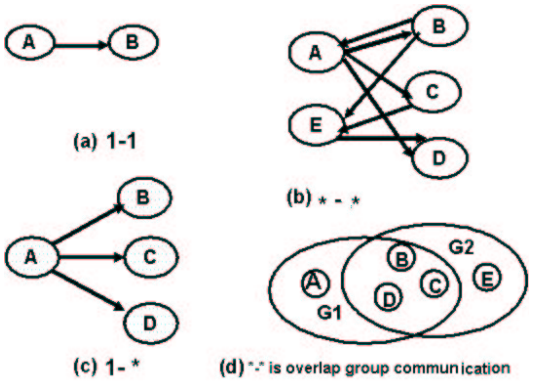

Figure 1.1: Communication Paradigm

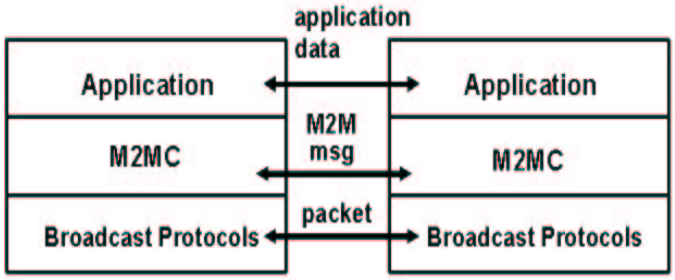

Figure 1.2: M2MC

processes to join and leave the groups. These middleware protocols broadcast messages to all nearby devices using some underlying reliable broadcast protocols.

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. Instead all the devices, which ever ones happened to be present in the changing set of proximal devices, act in concert to run the application.

1.2 Motivation

We have developed an application using our M2MC APIs called Threaded chat application. In this section we describe the 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 [1]. See [2] for a comprehensive survey of total ordering protocols.

As shown in Fig 1.3, 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 [3] or even when synchronized global clocks [4] are assumed.

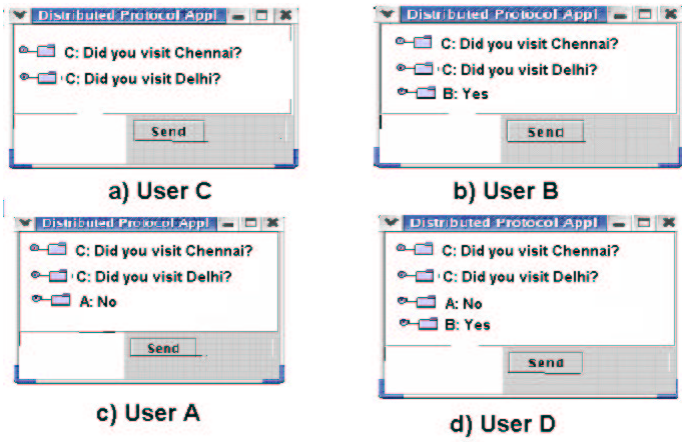

Figure 1.3: Chat Application

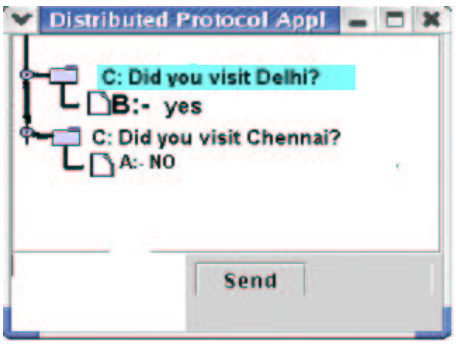

Figure 1.4: Threaded Chat Application

In contrast to the above, consider a threaded chat application [5] that lets users communicate in a **message–response** form as shown in Fig 1.4. 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.

Although the paper [5] 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 messages *'Did you visit Delhi?'* and its response *'Yes'* and sends this information to the group. The MOP at D, upon receipt of these responses, orders the messages appropriately and unambiguously, as shown in Fig 1.4. 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 0rders these messages and displays on the screen.

1.3 Objectives of the work

The objective of the present work is to conceive, design the components of M2MC and implement them as Java APIs for developing applications like threaded chat application, multiplayer games etc. The detailed scope of the work as follows:

- Conceiving the components of M2MC.
- Designing the architecture of M2MC.
- Implementing it as Java APIs for application development.
- Developing threaded chat application using M2MC.

1.4 Layout of the Report

The layout of the report is as follows:

- First chapter starts with introduction, followed by motivation and objectives of work.
- Second chapter reviews literature review one existing middleware for group communication.
- Third chapter presents the architecture and components of M2MC.
- Fourth chapter discusses Message Ordering Protocol and its algorithms.
- Fifth chapter presents Group Join/Leave Protocols
- Sixth chapter presents Member Synchronization Protocols.
- Seventh chapter presents Java implementation of M2MC
- Eight chapter discusses Threaded chat application development.
- Ninth chapter concludes the report.

Chapter 2

Literature Survey

2.1 Introduction

We have surveyed the middleware for one to one communications and group communications. We present here the one to one middlewares CORBA, RMI, group communication middleware like Amoeba and Anhinga. In this chapter we present the problems in using them for developing truly distributed group applications. We have also surveyed the multicasting protocols and we present some of them.

2.2 Existing middleware for point to point communication

CORBA and RMI are some of the many middleware concepts that exist in the wired network. The two mechanisms allow a client to access a remote object present in different machine and address space transparently. These middleware systems are build for unicast method invocation over TCP/IP and hence their original incarnations do not support muliticasting or group services.

2.2.1 CORBA

CORBA [6] specifies a system which provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In this model clients request services from objects (which will also be called servers) through a

well-defined interface. This interface is specified in OMG IDL (Interface Definition Language). A client accesses an object by issuing a request to the object. The request is an event, and it carries information including an operation, the object reference of the service provider, and actual parameters. The object reference is an object name that defines an object reliably.

2.2.2 RMI

RMI [7]is a Java version of RPC. RMI support remote invocation on objects across Java virtual machines. RMI directly integrates a distributed object model into the Java Language. RMI also includes the dynamic downloading of stubs. Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism. CORBA is another object-oriented RPC mechanism. CORBA differs from Java RMI in a number of ways:

1. CORBA is a language-independent standard.

2. CORBA includes many other mechanisms in its standard (such as a standard for TP monitors) none of which are part of Java RMI.

3. There is also no notion of an "object request broker" in Java RMI.

Java RMI has recently evolved toward becoming more compatible with CORBA. There is now a form of RMI called RMI/IIOP ("RMI over IIOP") that uses the Internet Inter-ORB Protocol (IIOP) of CORBA as the underlying protocol for RMI communication.

There are three processes that participate in supporting remote method invocation.

1. The Client is the process that is invoking a method on a remote object.

2. The Server is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.

3. The Object Registry is a name server that relates objects with names. Objects are registered with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

There are two kinds of classes in Java RMI. Remote class and Serializable class. Instances of the Remote class are accessable to remote client. The state of an object of a serializable class can be copied from one address space to another.

2.2.2.1 Not suitable for group communication

RMI, CORBA etc are suitable for point to point communication. Although group communication can be achieved by using point to point communication, they do not exploit the advantages of using multicasting or broadcasting protocols. Hence they are not suitable for group communication..

2.3 Existing middleware for group communication

2.3.1 Amoeba middleware for client server group communication

Amoeba is based on model in which all computing power is located in one or more processor pools. A processor pool contains substantial number of CPUs each with local memory and network connection. Availability of shared memory is not expected, but if available can be used for optimizing message passing by using memory to memory copying.

Amoeba consists of two basic pieces: a microkernel, which runs on every processor, and a collection of servers that provide most of the traditional operating system functionally. The Amoeba microkernel runs on all machines in the system. The same kernel can be used on pool processors, terminals etc. The microkernel has four primary functions:

- **Manage processes and threads**

  Amoeba supports the concept of a process and multiple threads of control within a single address space. A typical use for multiple threads might be for running file server etc.

- **Provide low-level memory management support**

  The kernel provides low level memory management. Threads can allocate and de allocate blocks of memory called segments. These can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs.

- **Support communication**

  The kernel supportsinterprocess communication. Two forms of communication provided are point to point and group communication. Point to point communication is based on client server model. Group communication allows a message to be sent from one source to multiple destinations. Software protocols provide reliable, fault-tolerant group communication to user processes in the presence of lost messages and other errors.

- **Handle low-level I-O** Devise drivers communicate with rest of the system by the standard request and reply messages.

Although Amoeba provides group communication, it assumes existence of central server and hence not suitable for truly distributed communication.

2.3.2 Anhinga middleware for peer2peer group communication

The Anhinga Project [8] 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). as shown in Fig.2.1. The architecture is built for mobile devices that are in the transmission range of each other. In this infrastructure, all the nodes that participate in the collaborative application will have and implementation of an object and interface.Each object is registered with M2MI layer as public object so that any node can invoke it (omni handle), restricted to a particular group (group handle) or unihandle object which means only one of the nodes can invoke it. So when a client calls an remote object after obtaining and interface of it, appropriate invocations happen at remote nodes based on their access control handles.

2.3.2.1 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

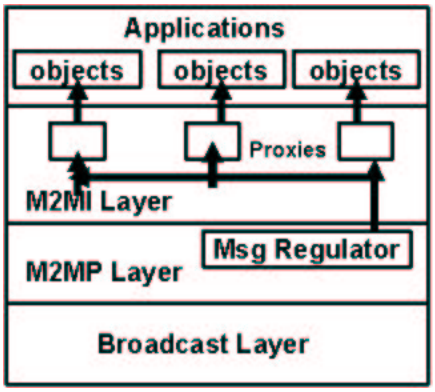

Figure 2.1: Anhinga Architecture

invocation, which are received and performed by many objects in many target devices simultaneously. The paper [8] describes it in detail.

2.3.2.2 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.

Since no routing is involved, every message is broadcasted. M2MP layer has filter implementations, and this layer will transfer data to M2MI layer only if it satisfied filter conditions. Hence this is a kind of message based or content based routing.

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.

2.4 Multicasting protocols:

Multicasting is the transmission of datagrams to a group of hosts identified by a single destination address. It is intended for group-oriented computing applications that are characterized by the close collaboration of teams with requirements for audio and video conferencing and sharing of test and images. Maintaining group membership information and efficiently delivering the multicast packets to all members is challenging in *adhoc* networks. The nodes in these networks consist of a dynamic collection of nodes with

sometimes rapidly changing multi-hop topologies that are composed of relatively low-bandwidth wireless links. Since each node has a limited transmission range, a source-to-destination path could pass through several intermediate nodes. The network topology can change randomly and rapidly, at unpredictable times. Most of the mobile devices are powered by batteries, thus routing protocols must limit the amount of control information that is passed between nodes.

The existing multicast protocols can be can classify into tree-based multicast protocols, mesh-based and flooding based protocols. Tree-based multicast protocols construct a logical multicast tree with the members of the group forming nodes of the tree. As the tree is a cycle free data structure, no alternate paths exists between any two adjacent nodes. These are not robust but less overhead to construct and maintain the tree. The following are two tree-based protocols.

2.4.1 Ad hoc Multicast Routing protocol utilizing Increasing id numbers(AMRIS)

AMRIS is a tree-based multicast scheme, hence uses a shared tree for multicast data forwarding. Each node in the multicast session is assigned a multicast session member id(msmid). The key idea in using msmid is to denote the logical height of the node at that particular instant in the shared multicast tree.Theprotocol has two mechanisms, they are Tree Maintenance and Tree Initialization. The nodes interested in joining the multicast group sends periodic beacon signals to find out the existence of a neighbour. Initialy a source node with id sid broadcasts a NEW SESSION packet. The relation between Sid and msm id is that msm ids increase in value as they radiate away from Sid. The neighbors upon receiving this packet calculate their own msmids and re broadcast the packet. The msmids may not be consecutive.Information derived from the NEW Session manage is kept in the neighbor status table for some t1 seconds. A node joins a session by determining its parent, i.e the neighboring nodes that have smaller msm ids. A unicast join is then sent to potential parent, i.e Join Req is sent and if the parent is in the delivery tree then it sends a Join ACK message, or that node will try to locate a potential parent for it. And a join ack is sent back through the reverse path, thereby drafting a branch to the node. The node receives all data through this branch. If no parent is found by

unicast request to neighboring nodes, then a broadcast message is sent to locate a parent. If node is unable to locate a parent then a BR (Branch Reconstruction) process is carried out. The Tree maintenance mechanism operates in parallel in the back ground. When some link failures occurs, the node with higher msm id i.e the child is responsible for reconstruction of link. BR has two sub routines, BR1 and BR2, where BR1 is executed when the node has neighboring potential parent and BR2 when no neighboring potential parent is present. In BR1 a node send a unicast message to neighbor parent, and the node sends a JOIN ACK, this process happens if the neighboring parent is no the multicast tree or else it sends JOIN NACK. Now BR2 is executed where a broadcast message is sent, and there is a restriction on the number of hoops. If a node succeeds to find a parent then it receives a JOIN-CONF.

2.4.2 Adhoc Multicast Routing Protocol

This protocol uses shared multicast tree approach. It runs over an underlying unicast channel. It allows dynamic core migration based on group membership and network configuration. The disadvantage of this protocol is it is vulnerable to core node. It requires a underlying routing protocol in addition to tree based multicasting mechanism.

The protocol assumes the existence of an underlying unicast routing protocol that can be utilized for unicast IP communication between neighboring tree nodes. The actual path followed may the two direction of unicast tunnel connecting neighboring group members may be different. It is not required for allnetwork nodes support AMRoute or any other IP Multicast Protocol. All group members must be capable of processing IP in IP encapsulation. No group members need have more than one interface or act as unicast routers. However, at least one member must be capable of being AMRouter.

The protocol creates per group multicast distribution tree using unicast tunnels connecting group members. The protocol has two main components:mesh creation and tree creation. There are three types of nodes , noncore node, core node and nongroup member. Only the logical core node intiates mesh and tree creation. `Bi-directional` links are created between a pair of group members that are close together and thus forming a mesh structure. Using a subset of the available mesh links protocol periodically creates a multicast distribution tree. The protocol works as follows. Each group has atleast one logical core that maintains the multicast structure and members. During the start

each member assumes that theyare the core of their own group, and the group size is one. Now the core periodically floods a `join request` to discover other mesh segments. When a node receives the message from a core of the same group but from a different mesh segment, it replies with a `join-ack`, and marks it as its mesh neighbor. Once the mesh is created each core sends a `Tree-create` packet periodically to its mesh neighbors for building the shared multicast tree. Members forward the non duplicate create tree packets to others, which they have received from one of their mesh links. If a duplicate tree create is received, then the node receiving the packet send a `tree-create-nak`, alon the incoming link. The nodes wishing to leave the group send a `join-nak` to their neighbors, and they do not forward any data packets for the group. The AMRoute prtocol uses virtual mesh links to create the shared multicast tree, so as long as routes between tree members exist through mesh links, the tree need not be re adhusted when there is a network topology change. So processing overhead is incurred only by the nodes that form the tree. Since there are loops present a non optimal tree is created when there is node mobility. Some features of the protocol

1. There is a channel overhead when sending the `Tree Create NAK` is used, to suppress the duplicates , but this would have been avoided by ignoring the duplicates.

2. Join NAK, also increases the processing and channel overhead. A more efficient solution could be using a time out procedure, i.e, if a node wishes to leave the group issues no more join requests; as a result the source or the logical core can remove this node from its routing table.

3. The protocol entirely depends on the underlying unicast protocol. If the unicast protocol fails then this protocol also fails.

The following protocol is a mesh-based multicast protocol. The protocol constructs a mesh with group members. This is robust protocol with possibly more than one path between any two group members.

2.4.2.1 On Demand Multicast Routing Protocol

On-Demand Multicasting Routing Protocol(ODMRP) is a mesh-based multicast scheme and uses a forwarding group concept. A subset of nodes forwards the multicast packets

via scoped flooding. The protocol dynamically build routes and maintain multicast group membership.

1. Route and Membership creation

   When a multicast source has packets to send it floods the network with an advertisement packet plus data piggybacked to it. This packet called JOIN QUERY is periodically flooded to refresh or update the route and membership information. When an intermediate node say X receives the packet from Y it stores the source ID into routing table and rebroadcasts it to its neighbors. If one of the neighbours is multicast receiver then it creates a JOIN REPLY and broadcasts to its neighbours (X is one among its neighbours). When X receives JOIN REPLY message it sets the FGFLAG (Forwarding Group Flag) and broadcasts its own JOIN REPLY. In this way node Y (which has sent JOIN QUERY) to X also sets FGFLAG and broadcasts JOIN REPLY. This is continued unpto the source node. The intermediate nodes which perform forwarding to their neighbors are called "forwarding group".

2. Example

   Consider the following figure for demonstration of the protocol. Nodes S1 and S2 are multicast sources, and nodes R1, R2, and R3 are multicast receivers. Nodes R2 and R3 send their JOIN REPLIES to both S1 and S2 via I2. R1 sends its JOIN REPLY to S1 via I1 and to S2 via I2. When receivers send their JOIN REPLIES to I1, I2 , they update their tables,sets FGFLAG set and builds its own JOINREPLY since there is a next node ID entry in JOIN REPLY received from R1 and R2 respectively. The JOIN REPLY built by I1 has an entry for sender S1 but not for S2because the next node ID for S2 in the received JOIN REPLY is not I1. The JOIN REPLY built by I2 will have and entry for both S1 and S2. In this way routing tables are setup.

3. Data Forwarding

   After group establishments and route constructions process by constructing forwarding tables, a source can multicast to group of receivers through intermediate forwarding nodes. An intermediate node forwards it only when it is not duplicate and the setting of the FGFLAG for the multicast group has not expired.This minimizes the traffic overhead and prevents sending packtes through expired routes. In

ODMRP, no explicit control packets need to be sent to join or leave the group. If a multicast source wants to leave the group it simply stops sending JOINQUERY , similarly the receiver does not JOINREPLY if it is no longer interested in receiving the message from that source and intermediate nodes will be demoted to nonforwarding status.

The flooding multicast protocols are best effort multicast protocols.

2.4.3 Adaptive Flooding

In adaptive flooding protocol, nodes dyanamically switch flooding mechanisms based on their perspective of network conditions. The relative velocity of nodes could be one such conditions. The flooding mechanisms discussed here are scoped flooding, hyper flooding.

1. Scoped Flooding

   The basic principle behind Scoped Flooding is to reduce re-broadcasts to avoid collisions and minimize overhead. Scoped Flooding is suitable for constrained mobility environments (for eg. in a conference scenario) where nodes do not move much and plain flooding will yield unnecessary re-broadcast. The work in bibtexRef shows that re-broadcast can provide between `0-61%` additional converage over what was already convered by a previous transmission. The coverage area reduces drastically to 0.05

   Different heuristics can be used in deciding whether to rebroadcast a packet. In scoped flooding implementation, each node periodically transmits *hello* messages which also contain the node's neighbor list. Nodes use hello message to update their own neighbor list and add received lists to their neighbor list table. When a node receives a broadcast, it compares the neighbor list of the transmitting node with its own neighbor list. If the receiving node's neighbor list is a subset of the transmitting node's neighbor list, then it does not rebroadcast the packet.

2. Hyper Flooding

   Nodes in hyper flooding mode periodically transmit hello messages. When a neighbor receives a hello message, it adds the hello message originator to its neighbor list(it the list does not already contain that node). Similarly to plain flooding,

when a node receives a data packet, it simply re-broadcasts the packet and also queues it in its packet cache. Additionally, re-broadcasts are triggered by two other events: receiving a packet from a node which is not in the current neighbor list or receiving a hello message from a new node. In these cases, nodes transmit all packets in their cache. The rationale behind re-broadcasts is that "newly acquired" nodes could have possibly missed the original flooding wave on account of its mobility. This increases overall reliablity by ensuring that new nodes entering the transmission region of a node receive data packets which they otherwise would have missed. Nodes periodically purge their packet cache to prevent, excess re-flooding of the packets.

These protocols is not suitable to support group communications because

1. Atomicity

   The protocol does not guarentee that either all nodes receive the packe t or none of them receive it, because the responsibility of forwarding the data is with not with the source node but with intermediate forwarding nodes.

2. Reliablility

   The nodes miss packets temporarily when the forwarding nodes move away. These protocols do not provide 100% guarentee that all the nodes of group receive data. Hence these are best effort service protocols.

3. Synchrony

   If a source S1 multicasts a message before another source S2 there is no guarentee that every member of the group receive the message from S1 first and then from S2.

2.5 Issues in Group Communication

2.5.1 Group Communication

A group is a collection of process or objects that act together in some system as specified by the application. The key property that all groups have is that when a message is sent to the group, all members of the group receive it. Groups are dynamic. New groups can be created and old groups can be destroyed. A node can join a group or leave one. A node

can be a member of several groups at the same time. Hence mechanisms are needed for managing groups and group membership. The purpose of introducing groups is to allow nodes to deal with collections of process as a single abstraction. Thus a node can send a message to a group of nodes without having to know how many there are or where they are, which may change from one call to next. The following are the design issues in wired network

2.5.2 Issues

1. Closed Groups versus Open groups

   Closed groups are those in which only members are allowed to send messages to other members in the group. In constast open groups are those that allow non members to send messages to members of the group. Closed groups are typically used for parallel processing. For example, a collection of process working together to play a game or developing a project without disturbance from nodes outside the group.

2. Peer Groups versus Hierarchial groups

   In peer groups all members have equal privilages. In hierarchial groups one or some of the nodes will coordinator or boss. The peer groups are symmetric but decision making is complicated. To decide anything voting has to be taken, incurring some delay and overhead. In hierarchial groups loss of coordinator brings entire group to halt.

3. Group Membership

   One approach is to have a centralised group server and alien nodes can register with the server. But if a mobile node hosting the server moves away then group collapses. Other approach is that nonmembers sending joining request to every memeber of the group or majority of the group.

   The other issues are that joining and leaving of the group have to be synchronous with messages being sent. In other words, starting at the instant that a process or node joined a group it must receive all messages sent to the group. Similarly as soon as the process has left a group, it must not receive any messages from the group.

4. Group Addressing

In order to send a message to a particular group, application process must have some way of specifying which group it means. One way is to give each group a unique address and if multicast support is there then group address will be same as multicast address. Another method of adressing a group is to require the sende to provide explicit list of all destinations. When this method is used, the parameter in the call to send that specifies the destination is a pointer to a list of addresses. This method has the drawback of lack of transparency of group members for the sender. The method is message based addressing in which a message is broadcasted and only some the receiver interested in the message pick the information and others filter it as noise.

5. Send and Receive primitives

   The existing RPC, RMI or CORBA middleware systems support request/reply model. These are suited for communication models in which client send a request and server replies it. But when a message is sent to a group of nodes, the sender cannot wait until it receives replies from all the members of the group. Hence request/reply model does not suit group communications. There should be one_way model in which sender sends messages to the group and if required one of the members explicitly call the sender to acknowledge the message. Hence RPC *send* and *receive* routines but be altered to make them $one - way$ and suitable for group communication.

6. Atomicity

   Group atomicity means either all members of the group must receive the message or none of them should receive it.

7. Message Ordering

   To make group communication easy to understand and use, two properties are required. The first one is atomicity that ensures all nodes receive message. The second one is that all these nodes receive the message in the order they are issued. One way to solve this problem is to have global time ordering. Absolute timing may not be possible always and hence weaker forms like logical clock based on the principle of causality.

8. Overlapping Groups

A process can be a member of multiple groups at the same time. This can lead to a new kind of inconsistency. Consider the figure shown below. There are two groups, 1 and 2. Process A,B, and C are members of group1. Process B,C, and D are members of group2. Now suppose that process A and D send information with global time stamp, the order in which B and C receive information from A and D differs i.e. B receives A's message first and then D's. C receives D's first and then A. Although global time ordering is used, the culprit is lack of coordination amount various groups.

Chapter 3

Architecture and components of M2MC

3.1 Middleware Architecture

As shown in Fig 3.1 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).

3.1.1 Components of Middleware

The following are components of our middleware.

3.1.1.1 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 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 3.2). 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 [3] or causal ordering protocol [1] do not take into account the semantic relationship among messages and hence are inadequate for many distributed group communication applications.

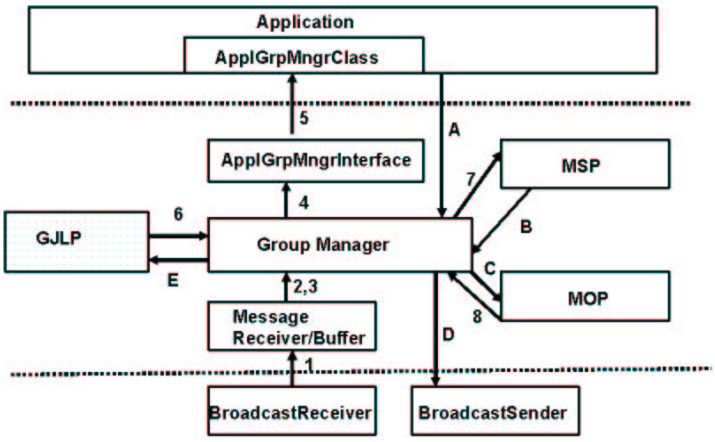

Figure 3.1: M2MC architecture

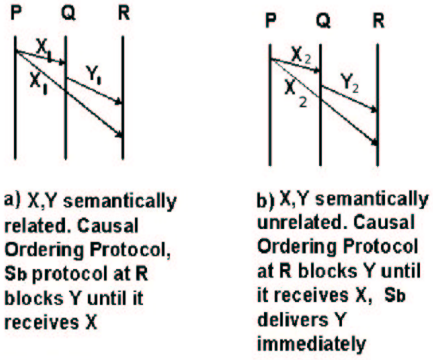

Figure 3.2: Illustrating S_b ordering

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).

3.1.1.2 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.

According to these protocols, when a process newly connects to broadcast network it advertises its presence to the group by sending the list of group information that it is aware of. Every process responds to the advertisement by sending the list of information about group that they are aware of. Every process in the network if finds any new groups information in these lists that it is not aware of, then it informs about them to the application.

3.1.1.3 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.

According to MSP, every process stores every message sent to the group and newly joined or rejoined process sends a request message containing the list of messages that

it has received. Every process updates their messages by delivering any new messages present in the list to the application. Any one of the processes(chosen randomly) present in the group by responding with the list of messages that the requested process missed. Only the requested process delivers the messages present in the list to the application.

3.1.1.4 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.

3.1.1.5 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.

3.1.1.6 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.

3.2 Middleware operations

3.2.0.7 For creating a new group:

The Application calls (step A in Fig 3.1) 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).

3.2.0.8 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.

3.2.0.9 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.

3.2.0.10 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.

3.2.0.11 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*.

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.

Chapter 4

Message Ordering Protocol

4.1 Message Ordering Protocol

Here, we present the specification and implementation details of Message Ordering Protocol. Group Join/Leave Protocol and Member Synchronization Protocols are presented in next chapters.

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.

4.1.1 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** $\mathbf{X} \xrightarrow{\neg S_b} \mathbf{Y}$.

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 4.1. The

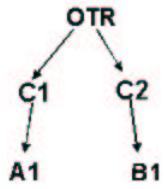

Figure 4.1: Ordering Tree

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.

4.1.2 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 4.1, 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:**

   $X \xrightarrow{S_b} Y \bigwedge Y \xrightarrow{S_b} Z \Longrightarrow X \underset{*}{\xrightarrow{S_b}} Z.$

   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 $\underset{*}{\xrightarrow{S_b}}$ to represent such transitive closure. It can be easily seen that the following also hold:

   - $X \underset{*}{\xrightarrow{S_b}} Y \bigwedge Y \xrightarrow{S_b} Z \Longrightarrow X \underset{*}{\xrightarrow{S_b}} Z$
   - $X \xrightarrow{S_b} Y \bigwedge Y \underset{*}{\xrightarrow{S_b}} Z \Longrightarrow X \underset{*}{\xrightarrow{S_b}} Z$
   - $X \underset{*}{\xrightarrow{S_b}} Y \bigwedge Y \underset{*}{\xrightarrow{S_b}} Z \Longrightarrow X \underset{*}{\xrightarrow{S_b}} Z$

4.1.3 Protocol Actions

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.

4.1.3.1 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 $< Msg >:< mid_c, mid_p, data >$.

4.1.3.2 State Diagram

The state diagram of the MOP is as shown in Fig 4.2. 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

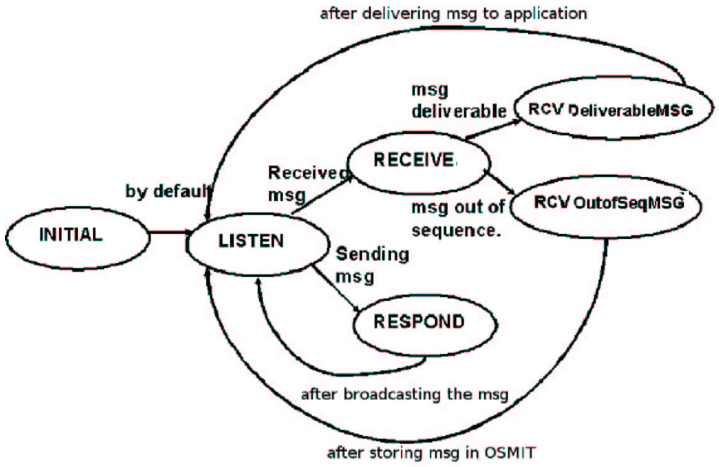

Figure 4.2: State Diagram of the protocol

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:

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.

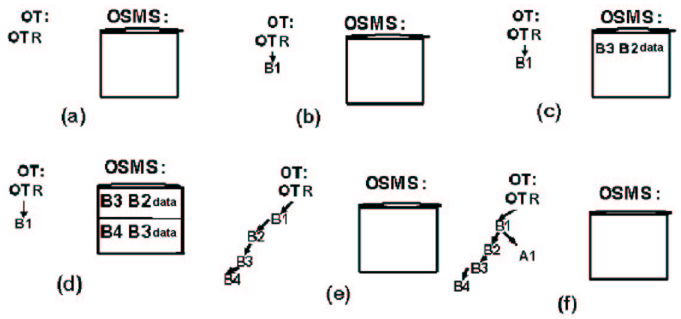

Figure 4.3: S_b protocol illustration

4.1.4 Protocol illustration

Consider a group formed by two processes with identities A and B respectively. We follow the Ordering Tree representation explained earlier, to show the semantic relationships among the messages exchanged between A and B. Recall that the root of the tree is a default node OTR and we assume OTR is semantically before every message sent to the group. The vertex of the tree represents message identity in the format $< pid, seqno >$.

Protocol Actions at process A are illustrated below:

1. In the initial state, the OT_A, OT_B at process A,B respectively contains root node OTR. Also the $OSMS_A$, $OSMS_B$ at process A,B respectively are empty. The sequence counters $seqno_A$, $seqno_B$ are set to zero. The process remains in LISTEN state until a message is received from the group or application responds to a previous message. The ordering tree is as shown in Fig 4.3a.

2. Process A on receiving a message $< B1, OTR, data >$ enters `RECEIVE` state. Since $m_p = OTR$ the process A goes to `RCVDelivarableMSG` state, saves B1 in OT_A as child node to OTR, and delivers the $data$ to the application. As the $OSMS_A$ is empty, it cannot find any messages m_r such that $(m_c \xrightarrow[*]{S_b} m_r)$ and goes back to LISTEN state. The data structures at process A are as shown in Fig 4.3b.

3. Process A on receiving a message $< B3, B2, data >$ enters RECEIVE state. Since $B2$ (mid_p field of the message), is not present in OT_A the protocol goes to `RCVOutSequenceMSG`.

Following the S_b protocol, the $< B3 >$, $< B2 >$, and *data* are saved in $OSMS_A$ as shown in Fig 4.3c.

4. Similarly on receiving $< B4, B3, data >$, the process A goes to RECEIVE state and then to `RCVOutSequenceMSG` because B3 is not in OL_{AB}. $B4$, $B3$, *data* are saved in OSM_A. The Fig 4.3d shows $OSMS_A$, OT_A.

5. On receiving $< B2, B1, data >$, the process goes to RECEIVE state and then to `RCVDeliverableMSG` state because $B1$ is present in OL_A. In this state, the protocol delivers *data* corresponding to received message and saves its identity $B2$ is saved in OL_A as child node of $B1$. Messages with identities B3, B4 are retrieved from $OSMS_A$ because B2$\xrightarrow{S_b}$B3 directly and B2$\xrightarrow[*]{S_b}$B4 transitively. The $< data >$ corresponding to messages with identities $B3, B4$ are delivered to the application in S_b order and identities B3, B4 are saved in OT_A as shown in Fig 4.3e.

6. If the application at process A wants to respond to a message having identity $B2$, then it goes to RESPOND STATE. It increments the sequence counter value to 1, receives *data* from the application and broadcasts the message in the format $< A1, B2, data >$. Since the message is a broadcast, process A on receiving its own message goes to `RCVDelivarableMSG` delivers *data* and updates OT_A as shown in(Fig 4.3f).

4.1.5 Correctness and liveness

We prove the correctness and liveness of the S_b protocol by using the semantic relationships among the messages as represented by the Ordering Tree (OT). Recall that by the transitive closure property of S_b order, the root of the OT is semantically before all messages sent to the group. $OTR \xrightarrow[*]{S_b} \{\forall messages \in OT\}$.

4.1.5.1 Correctness

Theorem 1: The S_b protocol preserves S_b ordering.

Explanation: Let $(OTR, A_1, A_2, A_3, A_4 \ldots A_n)$ be the identities of the messages in the OT along the path from root to any node A_n of the OT. As these messages are in S_b order, we need to prove that the S_b protocol delivers them to the application in the same

order.

Proof:

The proof is by induction on the number of messages n.

Base case:

For n = 1, the first message having identity A_1 sent to the group is not semantically before any other messages except OTR. Hence every process receiving it will deliver information corresponding to A_1 to the application. A_1 is represented as the child of OTR in OT.

If a process does not receive A_1 but receives a message A_2 such that $A_1 \xrightarrow[*]{S_b} A2$ then the process saves A_2 in the OSMS until it receives message A_1. The process after receiving A_1 will deliver information corresponding to A_1 to the application before delivering information of A_2. Hence the ordering is preserved.

Induction Hypothesis :

Assume that the S_b protocol preserves the ordering for n messages i.e for the messages $(OTR, A_1, A_2, A_3, A_4 \ldots A_n)$ in the ordering tree (OT).

Induction Step:

Suppose a member of the group sends a new message in the format $< A_{new}, mid_p, data >$ to the group.

- **If mid_p is node A_n**: Since A_n is present in the OT, the *data* corresponding to identity A_{new} is delivered to the application and A_{new} becomes child node of A_n in OT. Hence the protocol preserves the ordering for $n+1$ messages $(OTR, A_1, A_2, A_3, \ldots A_n, A_{new})$.

- **If mid_p is any node A_k in OT**. In this case also, the *data* corresponding to identity A_{new} is delivered to the application and A_{new} becomes child node of A_k in OT. Hence the protocol preserves the ordering for $n + 1$ messages.

- **If mid_p is not in OT**: In this case, A_{new} is saved in the OSMS until the process receives mid_p. Upon delivery of the message corresponding to mid_p, mid_p would be inserted into the OT. Now A_{new} is removed from the OSMS and also delivered to application. A_{new} then becomes child node of mid_p in OT. Hence the protocol preserves the ordering for $n + 1$ messages.

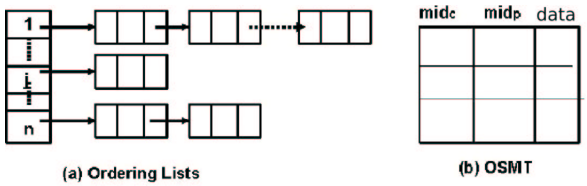

Figure 4.4: Data Structures

4.1.5.2 Liveness

Theorem 2: The S_b protocol is liveness preserving.

Explanation: Every message sent to the group will be eventually delivered to the applications at every process. We need to prove that no process will block a message indefinitely.

Proof:

Consider a message M for which **n** responses have been generated in the group. Consider the receipt of one of these responses, R, at process **i**.

- **If process i has delivered message M to the application**: In this case process **i** also delivers response R immediately, irrespective of other **n-1** responses.

- **If process i has not received message M**: In this case process **i** saves response R in the OSMS and waits for receipt of message M. Since the underlying broadcast medium is assumed to provide reliable message delivery, message M would be eventually delivered to process i. When process i receives message M, and subsequently delivers it, it traverses the OSMS and also delivers response R.

4.1.6 Protocol Implementation

Here, we discuss the protocol algorithm details and the corresponding time and space complexities.

The data structures required for S_b protocol at a process i are implemented as follows:

1. **Ordering Tree (OT_i):** We implement this as an array of linked lists, called Ordering Lists (OL_i), as shown in Fig 4.4a. The size of the array is equal to the number of processes present in the group. Each array element $OL_i[j]$ saves the starting address of the linked list corresponding to process j and the linked list saves the sequence

numbers of the messages received from process j. The data structure supports the following operations:

(a) **InsertInOL($seqno_j$)**: inserts $seqno_j$ in linked list starting at array element j.

(b) **IsPresentInOL($seqno_j$)**: searches for $seqno_j$ in linked list starting at array element j. If the $seqno_j$ is present then returns true else returns false.

2. **Out of Sequence Message Store($OSMS_i$):** We implement this as a 2-dimensional array of 3 columns each and some finite number of rows called Out of Sequence Message Table ($OSMT_i$) as shown in Fig 4.4b. Message identities of the messages that have arrived out of sequence are saved here. The process i on receiving a out of sequence message $< mid_c, mid_p, data >$ saves the identity of the parent message mid_c in the first column of the row, identity of the message mid_p in the second column and the $data$ in the third column.

   The data structure supports the following operations:

   (a) **InsertInOSMT($mid_c, mid_p, data$):**
   The operation uses the first empty row available from the top of the $OSMT_i$ table and inserts mid_c, mid_p, $data$ in the first, second, third columns of the row respectively.

   (b) **getRow(mid_p):**
   The operation searches in linear manner from the beginning of the $OSMT_i$ table and returns the index of row containing mid_p value in its second column. If there are multiple rows containing mid_p in their second column then it returns the first row that it encounters while searching from the beginning of the list.

   (c) **putOSMsgsInOL(mid_c):**
   The operation identifies all the rows of $OSMT_i$ containing messages for which mid_c is either directly or transitively semantically before them. The operation transfers the identities of these messages to OL_i and $data$ corresponding to these messages to application in S_b order. The rows containing these messages are marked empty for reusing them.

4.1.6.1 Protocol at process i

1. **In the initial state,**

   for $j = 1$ to n (where n is the number of members present in the group.) **do**

   $OL_i[j]$=NULL

   end for

   $seqno_i$=0

   The rows of $OSMS_i$ are marked empty.

2. **Process i on receiving a message:**

   On receiving a message $< mid_c, mid_p, data >$ from group, where $mid_c =< pid_c, seqno_c >$ and $mid_p =< pid_p, seqno_p >$

   if $mid_p \neq \emptyset$ and **IsPresentInOL**$(seqno_p)$ is false **then**

   InsertInOSMT$(mid_c, mid_p, data)$

   else

   InsertInOL$(seqno_c)$

   Deliver $data$ to the application

   putOSMsgsInOL(mid_c)

   end if

3. **Process i for responding to a message having identity mid_p**

   (a) $seqno_i = seqno_i + 1$;

   (b) $mid_i = < pid_i, seqno_i >$;

   (c) Broadcasts: $< mid_i, mid_p, data >$

4.1.6.2 Ordering List Operations

The structure of Ordering List is as shown in Fig 4.5. An array element $OL_i[k]$ saves the starting address of the linked list corresponding process k. Each node of the linked list contains there 3 fields. The first two fields called Low (L) and High (H) contains sequence numbers of the messages and the third field points to next node. The values L and H indicate that the process i has received all the messages having sequence numbers between L and H inclusive.

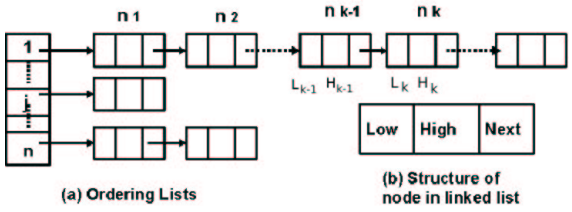

Figure 4.5: Ordering List Data structure

1. **IsPresentInOL**($seqno_j$)

   The operation searches for $seqno_j$ in linked list starting at $OL_i[j]$. If the $seqno_j$ is present then returns true else returns false.

Algorithm 1 IsPresentInOL($seqno_j$)

1: $S_j = OL_i[j]$ /* Starting address of linked list */
2: Scan the list and find the node n_k such that L_k is greatest number less than or equal to $seqno_j$
3: **if** $H_k \leq seqno_j$ **then**
4: return $true$
5: **else**
6: return $false$
7: **end if**

Time complexity: Scanning the list takes linear time, i.e., O(m) where m is the number of nodes present in the linked list.

2. **InsertInOL**($seqno_j$)

   The operation inserts $seqno_j$ in linked list starting at $OL_i[j]$. If process i receives messages with continuous sequence numbers starting from L and ending at H from a process j then these messages are represented in OL_i by storing only L and H values in a node of linked list starting at $OL_i[j]$. The linked list stores these values in the non decreasing order.

   So, to insert $seqno_j$, the linked list starting at $OL_i[j]$ is scanned to find the node n_k such that L_k is the least value greater than $seqno_j$ as shown in Fig 4.5a. Let n_{k-1} be the node preceding it. If $seqno_j$ is one more than H_{k-1} and one less than L_k then H_{k-1} is replaced by $seqno_j$ and the node n_k is deleted. Otherwise if $seqno_j$

is one more than H_{k-1} then H_{k-1} is replaced by $seqno_j$. Or else if $seqno_j$ is one less than L_k then L_k is replaced by $seqno_j$. If none of the above conditions satisfy then a new node n_{new} is created with L_{new} and H_{new} fields set to $seqno_j$ and n_{new} is inserted between n_{k-1} and n_k.

Algorithm 2 InsertInOL($seqno_j$)

1: 1. Scan the list and find the node n_k such that L_k is the least number greater than $seqno_j$.
2: **if** $H_{k-1} = seqno_j - 1$ and $L_k = seqno_j + 1$ **then**
3: 1.$H_{k-1} = H_k$
4: 2.Delete node k
5: **else if** $L_k = seqno_j + 1$ **then**
6: $L_k = seqno_j$
7: **else if** $H_{k-1} = seqno_j$-1 **then**
8: $H_k = seqno_j$
9: **else**
10: 1.Create a new node *new*
2.Assign $L_{new} = H_{new} = seqno_j$
3.Insert it between nodes n_{k-1} and n_k.
11: **end if**

Time complexity:

Scanning the linked list takes linear time, i.e., O(m) where m is the number of nodes present in linked list.

4.1.6.3 OSMT operations

The structure of $OSMT_i$ is shown in detail in Fig 4.6. We assume the table contains a maximum of **q** rows [1] for analyzing time complexity of the OSMT operations. Also for sake of clarity, we refer to a row of table $OSMT_i[k]$ as r_k and columns corresponding to row r_k i.e., $OSMT_i[k][0]$, $OSMT_i[k][1]$, $OSMT_i[k][2]$ as $r_k.mid_c$, $r_k.mid_p$, $r_k.data$ respectively.

1. **InsertInOSMT($mid_c, mid_p, data$):** As explained earlier the operation performs

[1] We assume q number rows of OSMT are sufficient for a process to save every message that arrives out of sequence.

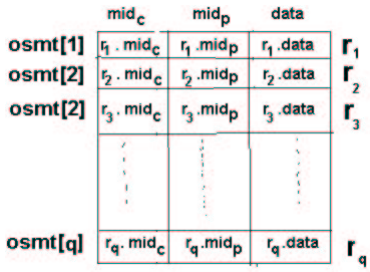

Figure 4.6: OSMT Data structure

linear search starting from the beginning of the table and searches until an empty row is found to insert the values mid_c,mid_p,$data$.

Time complexity:

Hence the InsertInOSMT operation takes O(p) time.

2. **getRow(mid_y):** As explained earlier the operation returns row r_x from OSMT such that $r_x.mid_p = mid_y$. If such row does not exist than it returns null. We assume the operation performs linear search from starting of the table to find the required row.

   Time complexity:

   Hence the above operation takes linear time, i.e., O(q) time.

3. **Operation: putOSMsgInOL(mid_x)** The OSMT contains collection of prospective edges of ordering tree OT that arrived out of S_b sequence in the form of rows of OSMT with each row containing parent and child message identities in columns mid_p, mid_c respectively. The operation starting from mid_x performs depth first search (DFS) on the contents of OSMT. It identifies the rows of OSMT that forms the edges of prospective sub tree of OT having mid_x as its root. During depth first search, every time a row is visited, the operation removes the row from OSMT, appends mid_c to OT and delivers $data$ to application. Hence the messages that arrive out of S_b order are delivered to the application in S_b order.

   Time complexity:

Searching the $OSMT_i$ and putting them into the OL_i takes quadratic time, i.e., $O((m+p)^2)$ (where m and p are as before).

Algorithm 3 putOSMsgInOL(mid_y)

1: **for** each row (r_x=getRow(mid_y))≠NULL **do**
2: Deliver r_x.data to application
3: InsertInOL($r_x.mid_c$)
4: putOSMsgInOL($r_x.mid_c$)
5: remove row r_x from OSMT
6: **end for**

This may be computed in detail as follows: Depth First Search algorithm for a tree with e edges takes O(e) time. Hence the DFS algorithm for $OSMT_i$ with a maximum of q rows takes O(q) time. To identify each edge of the tree and transferring it to OL_i, getRow operation at line 1 and InsertInOL operation in line 3 takes O(q),O(m) time respectively. Hence time taken by these operations together for each edge is O(m+q). Hence the operation putOSMsgInOL takes O(q(m+q)) time or $O((m+q)^2)$ time.

4.1.6.4 Complexity of S_b protocol

- **Time Complexity** The major operations of the protocol occur at a process while sending and receiving of messages.
  - Sending a message takes constant time as it involves only capturing mid_p from the application, incrementing the sequence number, and broadcasting the message.
  - On receiving a message, the **IsPresentInOL()** operation takes 0(m) time to check whether the message is deliverable to the application or not. If the message is not deliverable, then operation **InsertInOSMT()** takes linear time proportional to number of rows in $OSMT_i$ i.e., O(q). If the message is deliverable, then the operations performed are **InsertInOL(**$seqno_c$**)**, **putOSMsgInOL(**mid_c**)** and time complexity of each of these operations is 0(m),0($(m+q)^2$) respectively. (as discussed earlier). Hence the total time complexity of the S_b protocol is **0(**$(m+q)^2$**)**.
- **Space Complexity** The linked lists in the OL store only the first and last values of contiguous sequence numbers of messages received from a process.

- The **Space complexity for OL_i:**The linked lists in the OL store only the first and last values of contiguous sequence numbers of messages received from a process. The **best case** occurs when all the received messages have contiguous sequence numbers. Hence in the best case, each linked list contains only one node and size of array of linked lists is O(n) for a group with n processes. The **worst case** is when a process receives messages with alternate sequence numbers from every process. In this case, the number of nodes in each linked list is equal to the number of messages received from that process and the number of nodes present in the OL_i is equal to the number of messages received by the process from all members of the group. Hence in the worst case, the size of the OL is bounded only by the device's memory limits.
- The **Space Complexity for $OSMT_i$:**, the **best case** is when all messages are received in S_b order and the number of entries in the $OSMT_i$ is zero and the **worst case** is when a process does not receive the messages corresponding to the nodes closer the root of the OT_i and the number of entries in the $OSMT_i$ is bounded only by the number of rows allocated.

4.1.6.5 Java Implementation

The class diagram of the protocol and the java implementation of data structures are described chapter 7.

Chapter 5

Group Join/Leave Protocols

5.1 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.

5.1.1 Notations, Message Format and Data Structures

5.1.1.1 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.

5.1.1.2 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.

5.1.1.3 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**)

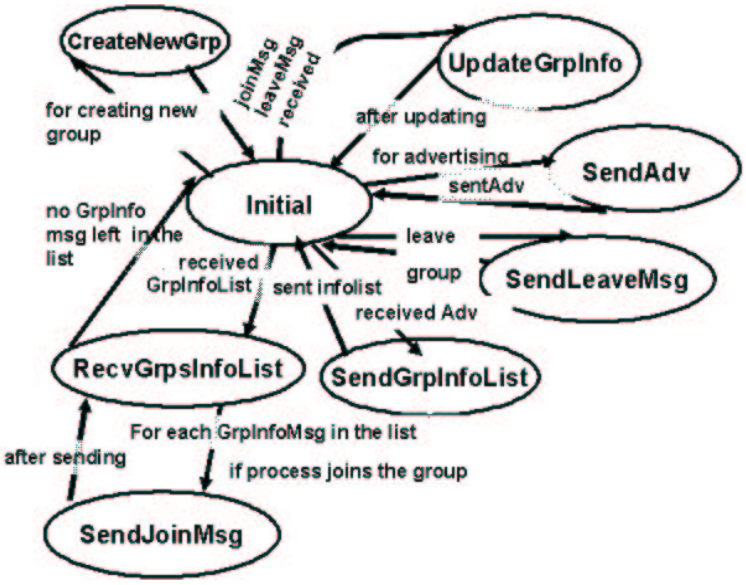

Figure 5.1: Group Join/Leave Protocol state diagram

5.1.2 Protocol Actions

The state diagram of the protocol at process **i** is as shown in the Fig 5.1. 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 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.

5.1.2.1 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 $grpInfo$.

5.1.3 Protocol Illustration

- Let a process A connects to broadcast network and creates a new group G1. Let process B connects to the network after sometime. After connecting to the network process B, following GJLP sends advertisement message. Upon receiving it process A sends *grpInfoList* containing information about group G1. Since B does not have information about any groups it does not send *groupInfoList*. Assuming process B joins the group, it sends *joinMsg* and upon receiving it both A,B update the members list of group G1.

- Let process C connects to the network. When C sends advertisement message, both A, B sends *grpsInfoList* containing information about group G1. Upon receiving any of these, process C delivers the group information to the user. Assuming C rejects the group invitation, it stores this information in its $GIIT$. Now C creates a new group G2 and broadcasts it.

- Assuming A, B have left the group temporarily, and process D connects to the group. When D sends the advertisement message, it receives *grpsInfoList* from C containing information about G2. If E joins G2 then it sends *joinMsg*.

- Suppose the processes A reconnects to the network then it sends advertisement message. Upon receiving it, process C, D sends *grpsInfoList* containing information about G2 and process A sends *grpsInfoList* containing information about G1. Hence every process in the network receives every group information.

5.1.4 Correctness and Liveness

5.1.4.1 Correctness

Every process present in the network receives information about every group that has been created so far, as long as one of the group members exists in the network. When process X newly connected to the network, according to GJLP it sends advertisement message, and every process in the network upon receiving it sends information about the groups that they are member of. Hence it receives information about every group. Suppose if every member of a group G goes off the network temporarily, then process X does not receive information about G. But when the any of the members of G reconnects to the network they send information about G after receiving their advertisement message. Hence process X will eventually gets information about G. If every member of a group G leaves the network permanently then information about the group G is lost.

5.1.4.2 Liveness

The protocol is deadlock free because, every process that runs GJLP, sends the *grpsInfoMsg* messages upon receipt of *advertisement* message. They do not wait for the arrival of *advertisement* message.

5.1.5 Protocol Implementation

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

Chapter 6

Member Synchronization Protocol

6.1 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.

6.1.1 Notations, Message Format and Data Structures

6.1.1.1 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.

6.1.1.2 Message Format

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

6.1.1.3 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.
- **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

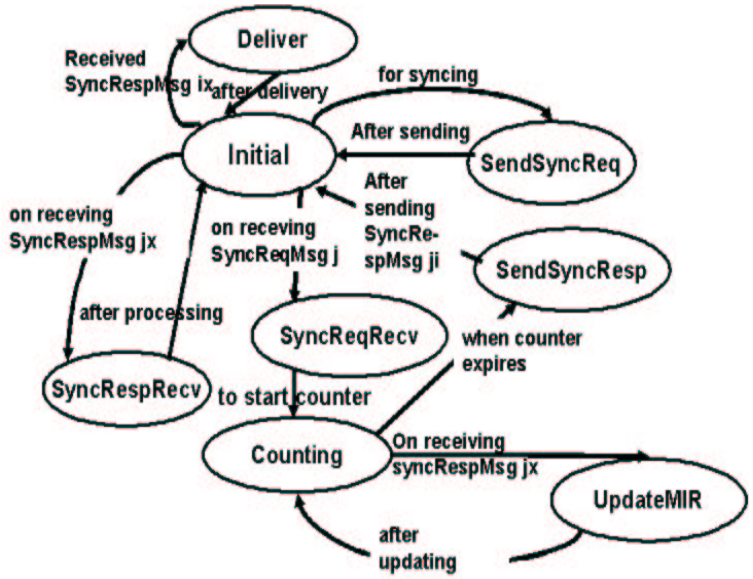

Figure 6.1: Member Synchronization Protocol

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 $SyncMsgId$.

6.1.2 Protocol Actions

The state diagram of the protocol at process **i** is as shown in fig6.1.

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 $SyncRespMsg_{jx}$·$SyncMsgList$.
   - 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 $SyncRespMsg_{jx}$·$SyncMsgId$, $SyncRespMsg_{jx}$·$SyncMsgList$ and $SyncRespMsg_{jx}$·$SyncMsgId \cdot gid$ 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

6.1.3 Protocol Illustration

Let A, B be members of the group and each of them having messages M1, M2 in their $GrpMsgList$ received from the group. Let process C which was member of the group and it went off the network temporarily. Let its $GrpMsgList$ contains M1 only. Now if processes C reconnects to the network and did not run MSP yet, and a new process D newly connects to the network. Assuming process D runs MSP before C, it sends *SyncReqMsg* to the group. Upon receiving *SyncReqMsg* process C, creates *MIR* with identity of M1 and A, B creates *MIR* with identities of M1, M2. Each of these start a random counter. Assuming counter at C expires it sends *SyncRespMsg* containing M1. On receiving it, process D sends M1 to application, and A,B deletes identity of M1 from their *MIR*. When the counter at A or B expires, A or B sends *SyncRespMsg* containing M2. Process D upon receiving it delivers M2 to the application.

6.1.4 Correctness and Liveness

6.1.4.1 Correctness

When a process runs MSP for a group, MSP ensures that every member of the group have same messages. This is because, when a newly joined or rejoined member runs MSP, it sends the messages that it is aware of, in *SyncReqMsg* and some of the members of the group whose random counters have expired, send *SyncRespMsg* containing group messages. Hence they exchange messages that they have and finally reach a synchronized state.

6.1.4.2 Liveness

The process running MSP does not wait for arrival of any message. It waits for only for the random counter to expire and that will eventually happen. Hence MSP does not suffer from deadlock problems.

6.1.5 Protocol Implementation

The class diagram of the protocol and the java implementation of data structures are described chapter4.

Chapter 7

Java Implementation of M2MC

7.1 Java Implementation of M2MC middleware Layer

7.1.1 System Environment

We have implemented the middleware protocols in Java language. We have used IP multicast for network layer protocols and the Multicast socket provided by the Java.io packet for Broadcasting over IP address. We discuss the implementation details of each protocol using their class diagrams.

7.1.2 Message Ordering Protocol

We have implemented the protocol as Java APIs. The class diagram of the protocol is as shown in Fig.7.1.

- **class:OutSeqMsgList** The class OutSeqMsgList implements the attributes and methods that support the operations of the data structure **Out of Sequence Message Store (OSMS)**. The store is implemented as linear linked list **osmMsgIdList** with each element of the list pointing to an object of class called **MsgId**. The **MsgId** stores mid_c and its corresponding mid_p of the message that arrived out of sequence.

  - void **insertInOsml(String midc, String midp)**
    - **Method Arguments:** The strings midc, midp represent the identity of a message and identity of its parent message respectively.
    - **Operation:**The function stores the identity of the message (that arrived violating S_b order) and its parent message identity in OSML. It creates an

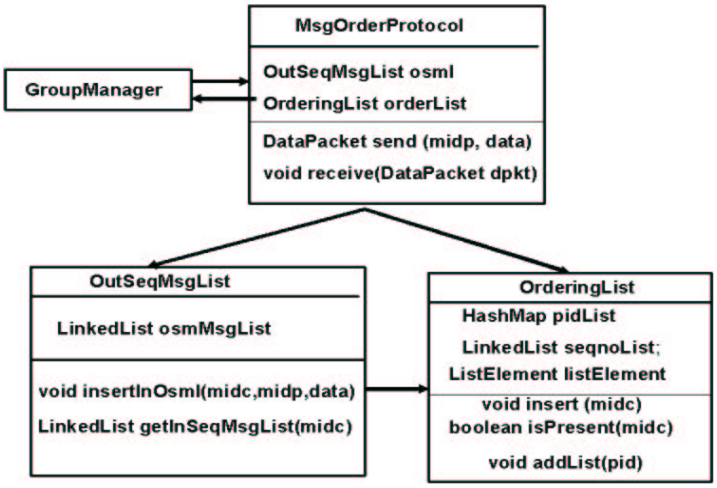

Figure 7.1: Message Ordering Protocol class diagram

object of class MsgId and stores midc, midp in it. The function appends the object of MsgID at the end osmMsgIdList.

* **Return type:** None.

– LinkedList **getInSeqMsgList(String** mid_c**)**

* **Method Arguments :** The String mid_c represents the identity of a message.

* **Operation:** The function extracts the objects of MsgId containing identities of messages from OSML that are semantically after mid_c and returns them in the form of elements of linked list called InSeqMsgList. The linked list OSML contains collection of prospective edges of ordering tree OT that arrived out of S_b sequence in the form of elements of linked list with each element storing the objects of MsgId. Each object of MsgID stores parent and child message identities in the fields mid_p and mid_c repectively.(Recall that OT is made up of nodes containing message identities and nodes representing parent and child messages identities are connected by edges) The method getInSeqMsgList starting from argument mid_c performs depth first search (DFS) on the elements of OSML. It identifies the edges of OSML that forms the edges of prospective sub tree of OT having mid_c as its root. During DFS, every time an element of the list is visited, the operation removes the object of MsgId pointed by the element and appends it to

linked list InSeqMsgList.

* **Return type**: returns object of linkedlist InSeqMsgList containing the identities of messages in S_b order.

– MsgId **getMsgIds(String** *mid***)**

* **Method Arguments**: The string *mid* is the identity of message.
* **Operation**: The function searches the elements of linked list OSML and returns the object of MsgId containing *mid* in its mid_c field.
* **Return type:** Object of MsgId containing identities of a message and its parent message.

- **class: Ordering List** The class Ordering List implements the attributes and methods that support the operations of data structure Ordering Tree (OT) as described I sec 2.2. The OT is actually implemented as collection of linked lists called seqnoList. If there are n members participating in group application, then n linked lists are maintained by the process such that one list per member of the group. The elements of linked list of member k stores the sequence numbers of the messages sent by member k. In order to save space, if process receives messages with contiguous sequence numbers say from low to high, from process k, then it stores only starting (low) and ending (high) sequence numbers of messages. The low and high values are stored in object of class ListElement that contains two integer fields (low and high) for storing these values. Each element in the linked list **seqnoList** points to the object of ListElement. A HashMap called pidList maps the identity of a process to the linked list seqnoLists corresponding to the process.

  – void **insert(String** mid_c**)**

  * **Method Arguments**: The argument mid_c represents identity of a message and it is combination of process identity pidc and sequence counter value ($seqno_c$) at process c when it sent the message.
  * **Operation**: The operation inserts the $seqno_c$ in linked list seqList. The operation get the starting address of the seqList by looking up the Java.util.HashMap *pidList* with key value pid_c. The seqList is scanned to find the element of list $listElement_k$ such that low_k value of the element is the least value

greater than $seqno_c$ as shown in figure. Let $listElement_{k-1}$ be the list element preceding the element $listElement_k$. If $seqno_c$ is one more than $high_{k-1}$ and one less than L_k then H_{k-1} is replaced by $seqno_c$ and the element $listElement_k$ is deleted. Otherwise if $seqno_c$ is one more than $high_{k-1}$ then $high_{k-1}$ is replaced by $seqno_c$. Or else if $seqno_c$ is one less than low_k then low_k is replaced by $seqno_j$. If none of the above conditions satisfy then a new object of class ListElement is created with low and $high$ fields set to $seqno_c$ and new element is inserted between the elements $listElement_k$ and $listElement_{k-1}$.

  - **Return type**: None.

- Boolean **isPresent(String** mid_c**)**
  - **Arguments:** The argument mid_c represents identity of a message and it is combination of process identity pid_c and sequence counter value ($seqno_c$) at process c when it sent the message.
  - **Operation:** The operation checks for the presence of sequence number $seqno_c$ in the linked list seqnoList. The operation get the starting address of the seqList by looking up the HashMap (pidList) with key value pid_c.
  - **Return Type**: The operation returns true if the $seqno_c$ is present in the linked list else returns false.

- void **addList(pid)**
  - **Arguments**: The argument pid represents the identity of a process.
  - **Operation**: The operation is called if a process with pid newly becomes member of group. The operation creates new object of LinkedList seqnoList for storing the sequence numbers of the messages sent by process pid. It also updates HaspMap pidList to include the mapping from pid to newly created linked list seqnoList.
  - **Return type:** None.

- **Class: MsgOrderProtocol** The object of class MsgOrderProtocol contains references to objects of classes OutSeqMsgList and OrderingList. For a given message identity, the methods of the class runs S_b protocol and decides whether the message X (and messages waiting in OSMS for the message X that is semantically before)

can be delivered to the application or it should be stored in OSMS before process did not receive the messages that are semantically before it. Also the MsgOrderProtocol contains methods for application to respond to a message sent to the group.

- void **send(String mid_p, String data)**
  - **Argument:** mid_p represent identity of the message and *data* represents the application data.
  - **Operation:** If the application wants to respond to a message having identity mid_p by sending *data* then this function is called. The method creates identity for the message and broadcasts the message in the format described in sec1.2.
  - **Return type:** None.
- void**receive(DataPacket dpkt)**
  - **Arguments:** The dpkt is object of class DataPacket. The class DataPacket contains fields mid_c, mid_p, *gid*, *data* representing the identity of the message, identity of its parent message, the identity of the group that it belongs to and the application information respectively.
  - **Operation:** The operation checks whether the application information present in the received dpkt is deliverable or not. It calls isPresent method of OrderingList class with argument mid_p. If it gets positive reply (true) from isPresent method (the method isPresent returns true if process already received message with identity mid_p.) then calls getInSeqMsgList(midc) of OutofSeqMsgList class to get the messages for which the received message is semantically before. It delivers the identities of these messages to Group Manager which in turn deliver the application information corresponding to these messages to the application.

7.1.3 Group Join/Leave Protocol

The class diagram of the protocol in the Fig7.2 shows salient attributes and methods of the class GrpJLProtocol.

7.1.3.1 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 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*.

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)*.

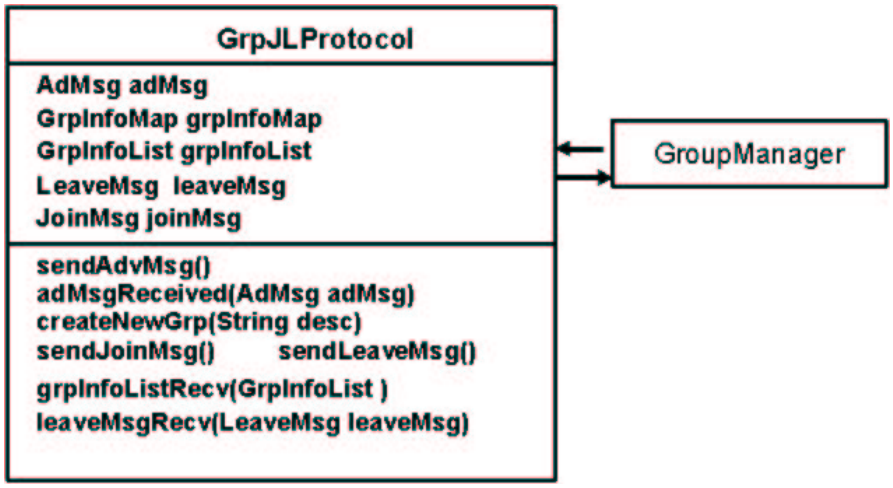

Figure 7.2: Group Join/Leave Protocol

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

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.

7.1.3.2 Member Synchronization Protocol

- **class: MemSyncProtocol** 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 $SyncMsgId$.

  1. **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.
  2. **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.
  3. **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.
  4. **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.

5. **sendSyncResp(syncMsgId)** gets the MIRList object 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.

- **class:Counter** The 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).

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

Chapter 8

Threaded Chat Application Development using M2MC

8.1 Case Study: Thread chat Application

As specified earlier, we can develop group applications like multiplayer games, chat applications etc with M2M middleware. In this section we describe a threaded chat application developed by using M2M middleware.

8.1.1 Motivation

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 [1]. See [2] for a comprehensive survey of total ordering protocols.

As shown in Fig 8.1, 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 [3] or even when synchronized global clocks [4] are assumed.

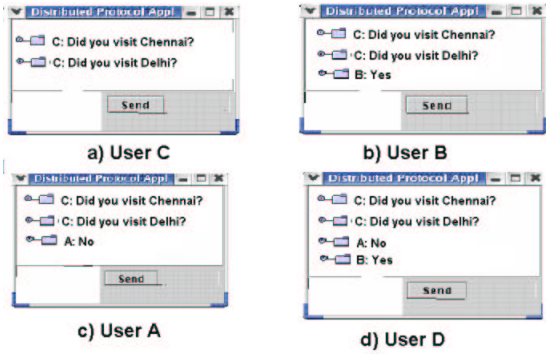

Figure 8.1: Chat Application

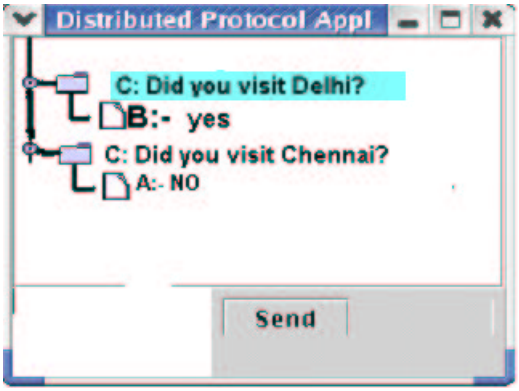

Figure 8.2: Threaded Chat Application

In contrast to the above, consider a threaded chat application [5] that lets users communicate in a **message-response** form as shown in Fig 8.2. 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.

8.2 Class Diagram

The class diagram of the application is as shown in the figure8.3.

8.2.1 class:GroupManager

The class **GroupManager** (Fig.8.4) of the M2M middleware provides the following method interfaces for the application developers. The class **GroupManager** maintains object of java.util.HashMap called *grpProtocolMap* for mapping identity of a group *gid* to its instances of classes MsgOrderProtocol and MsgSyncProtocol.

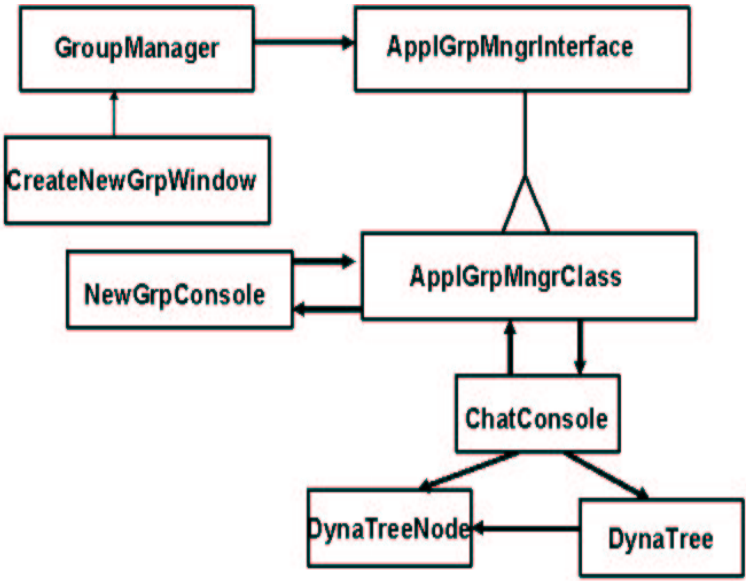

Figure 8.3: Threaded Chat Application Architecture

- 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 MsgSyncProtocol, 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.

- void **joinThisGroup(grpInfo)** is called if the process wants to join an existing group whose identity is grpInfo.gid. It creates instances of MsgOrderProtocol, MsgSyncProtocol, updates *grpProtocolMap* and calls *joinGrp(grpInfo)* of GrpJLProtocol class.

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

- 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 having identity mid_p. The method calls *send(midp,data)* of object of class MsgOrderProtocol corresponding to group with identity *gid*.

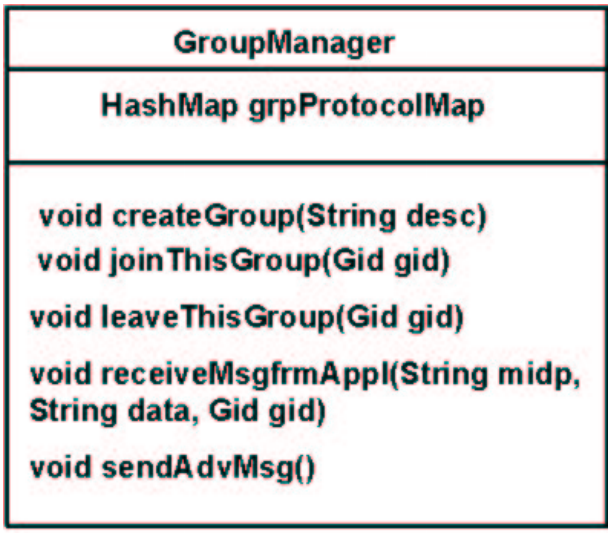

Figure 8.4: Group Manager class

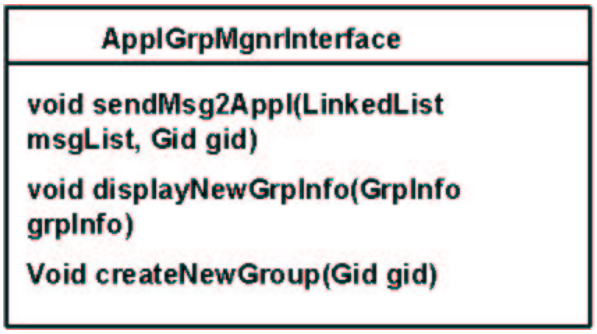

Figure 8.5: Interface for application developer

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

8.2.2 Interface: ApplGrpMgnrInterface

The interface ApplGrpMgnrInterface provides the following APIs as shown in Fig8.5. These APIs are called by the middleware software (methods of GroupManager). The application developer implements the methods provided in the interface based on application logic.

- 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. As discussed earlier each element of the linked list is an object to class MsgList containing the fields mid_c (identity of the message), mid_p (identity of the parent message) *data* (application information).

- void **displayNewGroupInfo(LinkedList grpInfoList)** The method is called by GroupManager, for sending the list of group identities and their descriptions. (Recall that when the process enters the network domain, it advertises its presence. The other processes in the network will respond to it by sending the identities and descriptions about the groups that they are aware of.)

- void **createdNewGroup(String gid)** The method 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.

8.3 Threaded Chat Application classes:

The classes ApplGrpMngrClass, GroupInfoWindow, ChatConsole, DynaTreeNode, DynaTree, represent the application logic.
The classes ChatConsole, DynaTreeNode, DynaTree, GroupInfoWindow provide GUI for the threaded chat application.

8.3.1 ChatConsole:

The ChatConsole creates the chat window (shown in Fig). The chat window contains main panel, text panel, buttons, and panel for displaying members. The main panel displays the messages in the form of tree structure such that a response to a message is connected by thread representing the parent−child relationship. The text panel is for entering the text message for broadcasting to the group. The following the methods of the class.

- void **createPanels()** The method calls the javax.swing library functions for creating various panels and buttons.

- void **actionPerformed()** The method implements the action listeners for **send** button such that when text is entered in the text panel after selecting one of the earlier messages and the button is pressed then, the action listener calls the sendMsg2Group() method of ApplGrpMngrClass for sending the text entered in the text panel and identity of the selected message to group members.

- void **display(Msg msg)** The argument msg is the object of class Msg that contains the fields, mid_c, mid_p, *data* representing the identities of message, its parent message and application information of message. The chat console represents the messages sent to the group in the form of tree which is the object of class called DynaTree and each message is the object of DynaTreeNode class. The method creates the object of DynaTreeNode (*child*) for application *data* received as argument, gets the object of DynaTreeNode (*parent*) corresponding to mid_p and the calls the function add2Tree() of DynaTreeNode class with arguments p, c for displaying on the chat window.

8.3.2 DynaTree

The DynaTree class contains the following methods:

- void **add2Tree(DynaTreeNode dynaParent, DynaTreeNode dynaChild)** The method represents the message contained in the object *dynaChild* on chat window as the child node of message contained in the *dynaParent* object.

- DynaTreeNode **getSelectedNode()** When user selects a message on the chat window, the actionListeners in class Console call this method to get the DynaTreeNode object of selected message.

8.3.3 DynaTreeNode

The object of the DynaTreeNode stores the chat message. It contains listeners for mouse events.

- **createNode(String msg)** The methods create a DynaTreeNode for given message string.

8.3.4 GroupInfoWindow

provides GUI for user to join new groups. It is as shown in the fig.

- void **createGroupInfoPanel()** The method creates the panel.
- void **showGroupdesc(String desc, String gid)** The method presents the group description and its identity on the group panel.
- void **actionPerformed()** The method is action listener and invoked when user decides to join the group by pressing the button. The method calls the method joinThisGroup(gid) of class ApplGrpMgnrClass with group identity parameter.

The ApplGrpMgnrClass implements the methods of ApplGrpMgnrInterface.

The following are methods of the class ApplGrpMngrClass that implements the following methods of the interface ApplGrpMngrInterface. Each group will have on unique chat window for displaying the messages sent to the group. The ApplGrpMngrClass creats maintains a HashMap to map the group identity gid to the object of ChatConsole class.

- void **sendMsg2Appl(LinkedList msgList, String gid)** The argument msgList is a linked list of messages with each element of the linked list pointing to the object of class Msg. The method displays the messages contained in msgList in the chat window corresponding to group with identity gid. The method calls display(msg) function of the chatConsole object with msg parameter for each element of the linked list. The display method displays the message on the window.
- void **displayNewGrpInfoList(LinkedList grpInfoList)** The argument is a linked list whose elements contain objects of class GrpInfo. The class GrpInfo contains fields group identity gid, group description desc. The method creates and object of GroupInfoWindow and calls the method show Grpdesc(desc, gid) for each element in the linked list.
- void **createdNewGroup(String gid)** The method creates new object of Console which in turn creates chat window for the group. The method also updates objects of HashMap that maps gid to object of Console.

- void joinThisGroup(String gid) The method calls the joinThisGroup(gid) method of GroupManager with group identity (gid) argument.

- void sendMsg2Grp(String mid_p, String data, Console console) The message is called by the chatConsole for sending a chat message *data* typed by the user in response to message with identity mid_p. The method finds the group identity (gid) of the console in which the message is typed using the HashMap and calls the method recvMsgfrmAppl(mid_p, *data*, *gid*).

The object of the class CreateNewGrpWindow (shown in Fig) contains the method for creating new group.

- JPanel **createPanel()** The method creates the panel for GUI.

- void **actionPerformed()** The method calls createGroup(String desc) of GroupMngr for creating new group.

Chapter 9

Summary and Conclusions

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. Message ordering protocols are key components for group communication systems. The most widely used message ordering protocols like total ordering, causal ordering protocols are not suitable for all group communication applications because they do not let the application explicitly specify the order among the messages.

We have defined a new ordering called S_b that orders the messages according to the semantic relationship among them as specified by the application and we described a protocol called MOP protocol that ensures all the receivers will receive the messages sent to the group in S_b order. We have explained the protocol actions with a state diagram. We have proved the correctness and liveness of the protocol and discussed the implementation issues and time, space complexity of protocol.

We have described the specification details and Java implementation details of these protocols. We have discussed Threaded Chat Application developed using M2MC.

Bibliography

[1] 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.

[2] 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.

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

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

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

[6] http://www.omg.org. *BASICS of CORBA*.

[7] http://java.sun.com. *Java RMI*.

[8] 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.

[9] Sung Ju Lee, William Su, and Mario Gerla. On-demand multicast routing protocol in multihop wireless mobile networks. *Mob. Netw. Appl.*, 7(6):441–453, 2002.

[10] Rajesh Talpade Jason Xie and Mingyan Liu Anthony Mcauley. Amroute: Ad hoc multicasting routing protocol. 2002.

[11] C-W.Wu and Y.Tay. Amris-a multicast protocol for ad-hoc wireless networks. 1999.

[12] . Andrew S. Tanenbaum. *Distributed Operating Systems.*

[13] Alan Kaminsky Hans-Peter Bischof and Joseph Binder. The anhinga project. a new framework for building secure collaborative systems in ad hoc network. 2003.

[14] Gene Tsudik Katia Obraczka. Multicast routing in ad hoc networks. 2000.

[15] Thomas Kunz and Ed Cheng. On-demand multicasting in ad-hoc networks: Comparing aodv and odmrp. 2002.

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

[17] C.Fidge. Timestamps in message passing systems that preserve the partial ordering. *Proceedings of 11th Australian Computer Science, page56-66*, pages 56–66, 1988.

[18] S.Mishra. F.Cristian, R.Debeijer. A performance companrision of asynchronous atomic broadcast protocols. *Distrib. Syst. Eng. J.1,4,177-201*, 1994.

[19] C.Fetzer F.Cristian. The timed asynchronous distributed system model. *IEEE Trans. Parall. Distrib. Syst. 10,6*, 1999.

[20] S.Mishra F.Cristian. The pinwheel asynchronous atomic broadcast protocols. *In Proceedings of 2nd International Symposium on Autonomous Decentralized Systems. IEEE Computer Society Press.*, 1995.

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

[22] M.Dasser. Tomp: A total ordering multicast protocol. *ACM Operat.Syst.Rev.26,1*, 1992.

[23] A.Schiper F.Pedone. Handling message semantics with generic broadcast protocols. *Distrib. Comput. 15,2,97-107*, 2002.

[24] A.Schiper. F.Pedone. Optimistic atomic broadcast: A pragmatic viewpoint. *Theor. Comput. Sci.291,79-101*, 2003.

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

Acknowledgements

I take this opportunity to express my sincere gratitude for **Prof. Sridhar Iyer** for his constant support and encouragement. His excellent guidance has been instrumental in making this project work a success.

I would like to thank members of the **Compaq lab and SIC313 lab** at KReSIT and **SIGNET** — the Special Interest Group in Networking, for their valuable suggestions and helpful discussions.

I would also like to thank my **family** and **friends**, who have been a source of encouragement and inspiration throughout the duration of the project.

Last but not the least, I would like to thank the entire KReSIT family for making my stay at IIT Bombay a memorable one.

Chaitanya Krishna Bhavanasi

I. I. T. Bombay

June 30^{th}, 2005