A Framework for Multiparty Communication Types

by

Chamath Indika Keppitiyagama

B.Sc. University of Colombo, 1997

M.Sc. University of British Columbia, 2000

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

July 2005

© Chamath Indika Keppitiyagama, 2005

Abstract

There are large number of communication paradigms such as multicast and anycast that are useful for distributed applications. The spectrum of such communication paradigms is larger than immediately apparent from the names used to identify these communication paradigms. None of these communication paradigms are universally available in the global Internet. To use these paradigms in applications programmers have to compose them using point-to-point communication or use third party modules that do so. Application level overlay networks have been used to implement some of these communication paradigms. These works indicate the vast design space behind the implementation of each of these multiparty communication paradigms over the wide area Internet.

Application programmers should be able to make use of communication paradigms independent of their implementations and the implementors of communication paradigms should be able to explore the design space of implementing them. The lack of three components inhibits achieving these goals; a naming system that can accommodate current and future communication paradigms, a common application programmer's interface (API), and a system to deploy the implementations of communication paradigms. This dissertation describes a framework named **MayaJala**, based on the novel notion of **multiparty communication types**, that addresses these issues; multiparty communication types are the precisely defined counterparts of multiparty communication paradigms. MayaJala consists of two main components; a communication type system and a middleware system.

The multiparty communication type system provides a mechanism to precisely identify communication paradigms. It also provides the ability to explore useful properties such as the equivalence of two communication types and conformance of one communication type to another. This allows applications to use different implementations of conforming communication types and not just different implementations of a single communication type. The multiparty communication type system also yields a common and simple interface sufficient for all the communication types.

The middleware allows dynamic deployment of implementations transparent to the applications and also provides common functionality required by these implementations.

The middleware provides support to implement communication types using application-level overlay networks. The middleware, together with the idea of multiparty communication types, facilitates the deployment of implementations of communication types without any coordination of the processes that participate in a session.

This work shows that it is possible to provide a naming system, and a simple and common API for multiparty communication paradigms without restricting or standardizing the set of such communication paradigms. This is achieved through the notion of multiparty communication types. This work also shows that multiparty communication types can be deployed without any coordination from the processes participating in a session. MayaJala provides these facilities with a minimum overhead to the applications that use it.

Contents

List of Tables

List of Figures

Acknowledgements

This work would not have been possible without the guidance and wisdom of my supervisor Norm Hutchinson. Norm is always willing to listen and he can easily put you at ease. I always came out of my weekly meetings with him more confident and cheerful than when I went in. I greatly appreciate his kind advice that shaped this work. Thank you, Norm, I have enjoyed being your student.

I would like to thank my thesis committee members, Mike Feeley and Alan Wagner for their valuable comments and guidance. Mike's ability to grasp the essence of a problem is legendary in the DSG lab. I greatly appreciate his input to this work as well as what I have learned from him during the 7 years I have been in UBC. Alan, guided me through my master's thesis work and provided valuable input to this thesis. He shaped the start of my research career. I greatly appreciate his comments and suggestions.

I would like to thank Will Evans for his guidance and support in completing this thesis. He patiently listened to me and gently suggested improvements to my thesis. I greatly appreciate his wisdom, insight, and patience.

DSG lab is a great place to work. I thank everybody in the DSG group for their help and support. A big thank you to Dima and Kan. Their friendship and support have given me the strength to go through some rough patches. I would like to thank Buck for the many interesting discussions that we had. I would also like to acknowledge the friendship and help of former DSG members, Ashley, Alex, Matt, Joon, Yvonne, Joseph, Kapil, and many others.

I wish to thank the fellow grads in the department for all the fun and excitement we shared over these years.

I would also like to thank the faculty and the staff of the department of Computer Science for their help and support on numerous occasions during my stay in the department. I acknowledge the support given to me by Richard Rosenberg, as my advisor, when I first came to the department. Joyce and Holly are always wiling to help and their help is greatly appreciated.

I would like to thank my parents for their continuous support, encouragement, and wisdom. They build the foundation of my life on which everything else I have achieved are

based on.

Finally, I would like to thank my wife, Prabha, for her love, patience, and support. This thesis would not have been possible without her support and encouragement.

CHAMATH INDIKA KEPPITIYAGAMA

The University of British Columbia
July 2005

Chapter 1

Introduction

Multiparty communication paradigms such as multicast, anycast, and manycast provide powerful abstractions to distributed applications. None of these communication paradigms are universally available in the global Internet. Often, the names used to identify multiparty communication paradigms do not precisely and unambiguously identify them. Multiparty communication types are the precisely-defined counterparts of multiparty communication paradigms. MayaJala[1] is a framework that consists of the multiparty communication type system and a middleware system that supports implementation, deployment, and use of multiparty communication types over the wide area Internet.

MayaJala is based on an assumption, and two extrapolations. We assume the following.

Multiparty communication paradigms provide useful and powerful abstractions to distributed-application programmers and make the programming task simpler.

From the existing work on multiparty communication paradigms we extrapolate the following.

The spectrum of multiparty communication paradigms is larger than apparent from the set of names used to identify them and extends beyond the known paradigms. There is a large design space behind each of these communication paradigms.

[1]We use the term MayaJala to denote the middleware itself and the complete framework that consists of both the middleware and the the multiparty communication type system.

Based on the above assumption and the extrapolations, we identify the following problems.

Ideally, application programmers should be able to use these paradigms independent of their implementation details and implementors should be able to explore the design space. However, this is hindered by the lack of three components; a naming system to identify the paradigms, a standard API, and a system to deploy the implementations.

In this work we address these issues.

We set the stage for the rest of the dissertation in this chapter. We start by presenting a motivating example that puts the above assumption in perspective. Then we briefly discuss the existing work that gave rise to the extrapolations, and the problems that we identified. Finally, we put forward the thesis and identify the contributions we make.

1.1 A motivating example

We start this discussion with a motivating example. Figure 1.1 shows the communication graph—a graph that depicts communication between processes—of a simple chat application that is deployed over the Internet. There is a server process (S) that moderates the chat session and all the other processes are chat clients. The server collects messages from each client and moderates the messages before sending them to all the clients. The clients can join and leave the session at any time. In Figure 1.1 an arrow from a process P_i to another process P_j indicates that process P_i sends messages to process P_j.

The graph is rather complicated and it depicts the communication at such a fine level of granularity that understanding the global communication structure is impossible. We need a higher level abstraction than a collection of simple point-to-point communications to glean useful information. However, it is only the point-to-point communication that is globally supported in the Internet. Even if we look at the source code of the programs of each process, we do not see the global picture—we can only observe the communication from the point of view of a single process. It is possible that each process is written by different programmers using different programming languages. The fact that the set of pro-

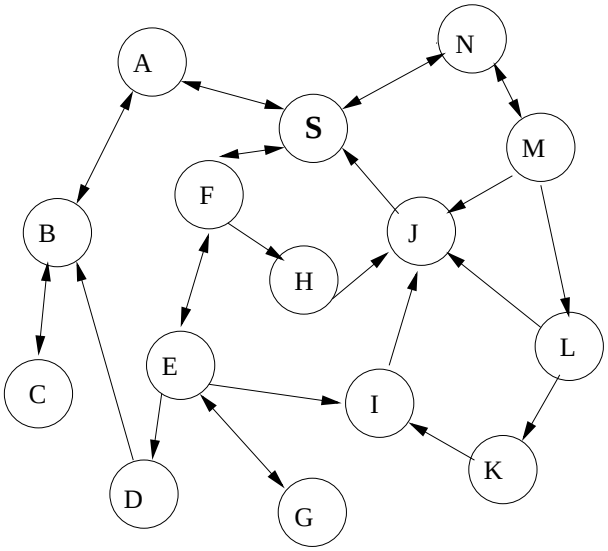

Figure 1.1: The communication graph.

cesses in this application can grow and shrink complicates the understanding of the graph even further. The communication graph changes with each change in the set of processes.

Figure 1.2, depicts the communication graph in Figure 1.1 at a higher level of abstraction. In this figure we identify two different patterns of communication. In one pattern, processes form a tree in the communication graph and send messages towards a single process, the server. In the second pattern one process sends messages down another tree to all the processes. This graph is also simplified for the sake of clarity. It is clear that the processes form *application level overlays* and forward messages. In reality there would be a control protocol associated with the construction and maintenance of the trees, which may include other processes that are not on the data path and may even have a different topology than the data path tree. Note that we may observe a different pattern of communication if we observe the communication graph at a different instance of time.

At this level of abstraction we get a better understanding of the communication structure and in the source code too the code corresponding to these two patterns can be isolated. This helps to understand the code better and also helps to debug the code. Kunz et al. [54] discuss the importance of identifying such communication patterns in debugging

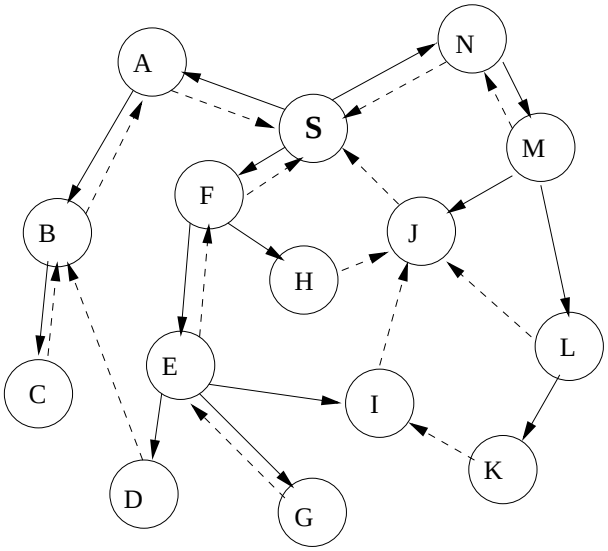

Figure 1.2: Two communication patterns.

parallel applications. However, note that as the set of processes grow and shrink the communication pattern changes. Therefore, we may not be able to talk about a constant pattern at all times.

Figure 1.3 depicts the communication at a even higher level of abstraction—we use two diagrams for clarity. At this level of abstraction we describe the communication in terms of two communication paradigms, multicast and gather. Multicast is a familiar communication paradigm, but gather is not a well known communication paradigm. However, similar communication paradigms have been discussed in the literature [19, 4]. In this particular case, multicast, the server sends messages to the "cloud" and all the other processes connected to the cloud receive the same message. The fact that all the receivers receive the same message is not apparent from the diagram and it is a property of the communication paradigm multicast. The cloud represents an instance of multicast. In the other diagram we have an instance of the communication paradigm gather. In this case the clients send the messages to the gather "cloud" and the messages are received by the server.

At this level of abstraction the programers do not have to worry about the intricate details of the communication patterns in implementing the paradigm. The actual commu-

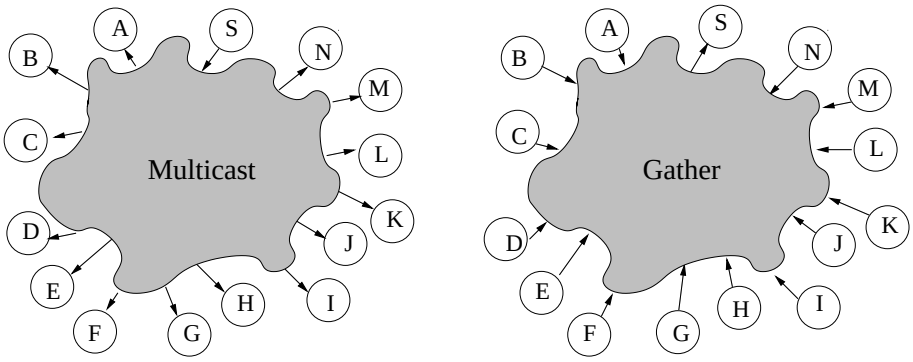

Figure 1.3: Two communication paradigms.

nication graph may change over time, but the programmers view is through the communication paradigm. Most importantly, we do not have to reason about the communication at the level of point-to-point communication. Gorlatch [39] equates send-receives to the "go-tos" of unstructured programs. In his article titled *Send-receive considered harmful* [39], he argues that collective communication is the suitable level of abstraction for parallel programs. When the communication code is written using low level abstractions like sends and receives mismatching mistakes can frequently occur. Pedersen et al. [74] discuss the problem of matching corresponding sends and receives in parallel applications to help the debugging processes. The importance of multiparty communication abstractions have been discussed extensively in the context of concurrent programming languages [49, 34, 37]. Francez et. al [37] stress the importance of hiding the low level details that implement communication patterns and present the Script abstraction mechanism for concurrent programming languages like CSP and Ada. The above example shows that these ideas are equally applicable to loosely coupled distributed applications as well.

Communication paradigms capture the essence of the communication independent of the actual communication pattern that implements the paradigm. Instead of the communication pattern in Figure 1.2, a communication pattern that sends a message to all the other processes directly can also be used to implement multicast. The communication paradigm multicast hides these details and modularizes the communication. Since multicast is not universally available in the global Internet, applications have to build some type of overlay

to implement multicast. This may be as trivial and inefficient as the sender itself replicating messages to every receiver.

1.2 The spectrum of communication paradigms

In the above example we used the term multicast to identify one of the communication paradigms used. Often, the term multicast does not identify a precise communication paradigm. In fact, multicast denotes a family of communication paradigms. IP multicast defined in RFC1112 [32] denotes a communication paradigm that allows multiple sources, that may not even be members of the group, to send messages. On the other hand Express multicast [44], another proposed architecture for IP multicast, only allows a single process to send messages. Overcast [47] is also a single source multicast implementation but it also provides some delivery guarantees. In contrast to IP multicast, which does not provide any ordering or delivery guarantees, the multicast paradigms ABCAST and CBCAST [16] provide strict ordering and delivery guarantees. This shows the spectrum of communication paradigms that are often simply referred to as multicast. Different applications have different requirements and an application that requires IP multicast semantics may not be able to use Express multicast without major modifications to the design. Therefore, it is important that we name these paradigms precisely.

Even though they are not as popular as multicast, there are other communication paradigms discussed in the literature. Anycast is another communication paradigm that has attracted attention because of its applicability to server selection applications. The exact semantics of anycast depends on different implementations and proposals. The semantics of application level anycast proposed by Bhattacharjee et al. [12] is different from IPv6 anycast [31]. Manycast [23] is a communication paradigm in which a message is delivered to a specified number of receivers. The communication paradigms concast [19] and gathercast [4] are somewhat similar to the gather communication paradigm we used in the above example. The communication primitive `MPI_gather()` defined in the Message Passing Interface (MPI) [65, 66] also has similarities to gather. The communication paradigms

somecast [96] is described with respect to a single application. Yet, it is easy to envisage a generalization of this paradigm.

We discuss these communication paradigms and their implementations in detail in Chapter 2. It is possible that there are variations of these communication paradigms as in multicast. We believe that the spectrum of communication paradigms are much larger than the ones discussed in the literature.

1.3 The design space of communication paradigms

There are several implementations of IP multicast, in terms of different routing protocols such as DVMRP [92], PIM-DM [33], PIM-SM [33], and CBT [6]. To the end user these routing protocols are transparent. These routing protocols address different issues in implementing IP multicast and are suitable for different situations. For example, PIM-DM is a routing protocol suitable when there is a large number of group members and PIM-SM is suitable when there are only a few widely dispersed group members. DVMRP does not scale well and PIM and CBT have been proposed to address the scalability issues. These protocols are the evidence of the large design space of IP multicast.

There is a large body of work that implements multicast using application level overlay networks. Application level multicast provides an alternative to IP multicast which is not deployed universally. Implementing multicast at the application level also provides more flexibility to the designers to explore the design space. These implementations explore ways to maximize bandwidth; minimize latency, link stress and stretch [27]; provide reliability; construct robust delivery trees; and improve other aspects that can, and should, be hidden from the application programmers that use these implementations. Even a seemingly mundane task like measuring the bandwidth of a link between two nodes, which is a common task in most overlay based multicast implementations, is a very complex task that warrants research in its own right [56].

The design space of other communication paradigms has not been explored to the extent of multicast. As discussed previously there are few proposed implementations of

anycast. Metz [67] describes challenges in implementing anycast. Carter et al.[21] presents several routing algorithms to implement manycast in mobile ad hoc networks. Catro et al. [23] implements manycast in different settings using a structured peer-to-peer overlay network. It is reasonable to assume that these not so well explored communication paradigms also have large design spaces as does multicast.

Note that the design space of collective operations in MPI over the wide area has been studied [52] as well as in other environments [10, 90, 97]. Squyres [85] also notes this vast design space. Some communication paradigms for distributed applications have similarities to the collective communication primitives in MPI (such as gather we used in the example). Therefore, the design space of the MPI collective communication primitives can also be taken as an indicator of the design space behind the communication paradigms for distributed applications.

1.4 Obstacles

Communication paradigms provide a powerful set of abstractions to distributed-application programmers. Ideally, application programmers should be able to use these communication abstractions in their applications independent of their implementations. This allows application programmers to concentrate on their programs while allowing the implementors of the communication paradigms to explore the design space. This also provides an opportunity to modularize the communication. This idea is reminiscent of the collective communication primitives in MPI [65, 66] and techniques such as Script [37] for concurrent programming languages that allow modularizing communication patterns.

We identify three main obstacles to use these communication paradigms in applications independent of their implementations. One obstacle is the lack of a method for application developers to specify the communication paradigm that they want to use and the implementors to specify what they implement. As we mentioned before, it is not sufficient to identify a communication paradigm as just multicast or anycast; these terms do not pinpoint the precise communication paradigms. Another obstacle is the lack of a com-

mon interface that all the implementations of a given communication paradigm adhere to. Standardizing an interface, as in MPI, is not practical because of the large number of communication paradigms possible. The third obstacle is the lack of a uniform method to deploy new implementations transparently to the applications. The components of today's Internet applications are often written independently and a component may communicate with one set of components in one session and a different set in another. For example, a video client may join one multicast session to receive a video stream from one server and may join another session to receive a stream from another server. Ideally there should be flexibility to use different multicast implementations for these two sessions; this requires transparent and dynamic deployment of the implementations.

We discuss these issues in detail in Chapter 3.

1.5 MayaJala

We provide a framework, named **MayaJala**, to overcome the above mentioned obstacles and to allow applications to use communication paradigms independent of their implementations. This framework includes an abstract model that allows a large number of communication paradigms to be precisely defined. The precisely defined communication paradigms are called **multiparty communication types**. The idea of multiparty communication paradigms is formally presented in Chapter 4. The model, in addition to yielding precise specifications, provides the ability to explore useful properties such as the equivalence of two communication types and conformance of one communication type to another. This allows applications to use different implementations of conforming communication types and not just the different implementations of a single communication type. From the model we derive a common and simple interface sufficient for all the communication types that can be described using this model. Finally, we provide a middleware that allows dynamic deployment of implementations transparent to the applications and also provides common functionality required by these implementations. The middleware provides support to implement communication types using application-level overlay networks.

1.6 Thesis

A broad class of multiparty-communication paradigms can be implemented by a common API and can be described by a formal model. Doing so simplifies an application's choice of communication library by syntactically de-coupling applications from communication libraries and by clarifying the semantic interdependence between the applications and libraries.

The communication type system precisely defines a broad range of communication paradigms; these are called communication types. A communication type clearly identifies the responsibilities of its implementations and the applications that use them. The communication type system also yields a common API for all communication types. The clear division of the responsibilities and the common API allow communication types to be implemented independent of the applications that use them. The MayaJala framework provides the support structure to implement and deploy communication types and facilitates the realization of the ideas developed with the communication type system in practice.

1.7 Contribution

There are four main contributions from this work.

1. The idea of multiparty communication types (discussed in Chapter 4).
2. A prototype model to define communication types (discussed in Chapter 4).
3. A framework, MayaJala, that supports implementation and deployment of communication types (discussed in Chapter 5).
4. A prototype implementation of MayaJala (discussed in Chapter 5).

Note that this work is not an attempt to provide a large number of communication types; nor it is an effort to provide optimized implementations of communication types. This work is about facilitating such efforts.

1.8 Road map

The rest of this thesis is organized as follows. In Chapter 2 we discuss some communication paradigms and their implementations to highlight the spectrum of communication paradigms and the design space behind them. In Chpater 3 we identify several unresolved issues that hinder the use of communication paradigms in applications independent of their implementations. In Chapter 4 we formally define the notion of multiparty communication type using a prototype model. The design and implementation of MayaJala is presented in Chapter 5. We evaluate the prototype implementation of MayaJala and some aspects of the notion of communication types in Chapter 6. Finally, in Chapter 7 we summarize the work, make conclusions, and discuss future work.

Chapter 2

Background

The purpose of this chapter is to show the breadth of the existing and possible communication paradigms and the vast design space in implementing these communication paradigms over the wide area Internet. First, we present some of the communication paradigms and several implementations. We discuss communication paradigms under several headings; multicast, anycast, manycast, gather, and other. We selected these headings solely for the purpose of presenting this material clearly and we do not present them as a precise categorization of all communication paradigms. One of our arguments in this thesis is that communication paradigms are not named precisely and we do not attempt to name them precisely at this juncture of the discussion; our solution to naming is presented in Chapter 4. Note that this collection of communication paradigms and implementations is not comprehensive, but we believe that it is adequate to highlight the possible communication paradigms and the design space. Finally, we discuss Message Passing Interface (MPI) that provides several collective communication primitives to parallel-application programmers. We pay special attention to MPI since it provided motivation for this work.

2.1 Multicast

Multicast is the best known multiparty communication paradigm. However, the exact meaning of multicast depends on the implementation and the context. The same term multicast

has different semantics in IP multicast, some application-level multicast implementations, and in group communication systems. We present the work related to the multicast family of communication paradigms under three headings.

1. IP multicast.
2. Overlay based multicast.
3. Multicast in group communication systems.

2.1.1 IP multicast

Multicast was introduced to the IP protocol suite by Deering [32] in the late 80's. IP multicast is described as a best effort service that delivers multicasted packets to a set of hosts identified by a single multicast group address (class D address). IP multicast provides "open group" semantics; that is, it does not require a host to be a member of a group to send a message to the group. The group address provides an indirection between the senders and the receivers. The groups are dynamic and the membership of the group is transparent to the group members (and to the non-group senders).

A host joins a multicast session by informing the router in its own network using Internet Group Management Protocol (IGMP). Routers use a multicast routing protocol to forward packets to the interested receivers. There are several routing protocols proposed for IP multicast, such as Distance Vector Multicast Routing Protocol (DVMRP) [92], Core Based Tree (CBT) [6] multicast routing protocol, and Protocol Independent Multicast (PIM) [33]. DVMRP is designed as an *interior gateway protocol* to be used within an autonomous system. DVMRP builds a sender specific multicast delivery tree based on the reverse path forwarding (RPF) algorithm. The first few packets reach all the routers along this tree. The routers that do not have members for the group send "prune" messages towards the tree and the delivery tree is truncated. In DVMRP the routers must keep per-source information and this is not scalable. CBT in contrast does not build source specific trees. There is a single multicast tree rooted at a *core* router. In CBT the core can be a bottleneck and PIM

allows both shortest path trees and shared trees to remedy this. PIM also has sparse mode operation which efficiently handles the case where there are only few senders and receivers in the multicast group. These different routing protocols show the design space behind IP multicast.

In RFC 3569 [13] Bhattacharyya describes the IP multicast, as described in RFC 1112, as **any-source multicast** (ASM) as opposed to **source-specific multicast** (SSM). He identifies three problems related to IP multicast; possibility for address collision, inability of the receivers to limit the sources, and inefficient handling of well known sources. Bhattacharyya [13] discusses source-specific multicast as a solution to these problems. SSM has multicast channels identified by source and group pairs. Bhattacharyya mentions that SSM was derived from the Express multicast channels proposed by Holbrook and Cheriton [44]. Note that Holbrook et al. also identify breaking of the normal billing model as a problem associated with IP multicast; the input-rate billing model does not account for the burden placed on ISPs by IP multicast. Express multicast channels are owned by a single source and ISPs can have a simpler billing model to charge the source. Since there is a channel per source, collisions in address allocation can be avoided. Also the subscribers can specify the source to receive from. Note that IGMPv3 also allows subscribers to normal IP multicast sessions to exclude sources.

2.1.2 Overlay based multicast

Currently there is no universal deployment of IP multicast. While the routers support IP multicast, deployment across autonomous systems requires support and peering arrangements between the ISPs. This is not a trivial task. There have been limited success in deploying IP multicast over MBone which connects IP multicast capable “islands” using overlay tunnels.

Recently a large body of research has emerged to support multicast without the support of the routers and ISPs by forming application level overlay networks. The flexibility to deploy different routing and tree construction algorithms without the support of the ISPs

has allowed researchers to explore the design space in implementing multicast. We discuss some of these works to emphasize on the large design space.

ALMI

Application Level Multicast Infrastructure (ALMI) [75] supports multi-sender multicast. ALMI constructs a Minimum Spanning Tree (MST) overlay over the end-hosts for a multicast session. Each multicast session has a centralized session controller. The session controller gathers application specific link metrics between the end-hosts and calculates the MST. To reduce the link monitoring, ALMI asks end-hosts to monitor only a subset of other end-hosts rather than requiring each end-host to monitor links to all the others. Session members can send and receive data on this MST and must forward them to the adjacent nodes.

To join an ALMI multicast session a new member must contact the session controller and get the ID of its parent on the MST and a set of nodes to monitor. The new member then sends a GRAFT message to the parent to connect to the MST. If a node gets to know that its parent node has departed from the session, then it must contact the session controller again to re-join the session.

ALMI by itself does not provide any means of communicating the identity of the session controller and it does not mandate any specific location for the session controller. However, the authors suggest to co-locate the session controller with any of the session members— specially with the session initializer.

Multicast using i^3

Lakshminarayanan et al. [57] introduce a multicast implementation based on the Internet Indirection Infrastructure (i^3) [87]. i^3 is implemented as an overlay network and it provides the decoupling between sending and receiving. In i^3, messages are sent to an identifier and receivers who are interested in receiving messages insert triggers for these identifiers. The i^3 infrastructure provides a set of servers that store triggers. The server responsible

for an identifier is found by routing a message over the Pastry [84] overlay to the identifier. A message sent by a sender to an identifier also arrives at the server responsible for the identifier and if it matches a trigger it is forwarded to the receivers.

Small scale multicast is implemented simply by all the group members inserting triggers with the same identifier, which acts as the group identifier. In i^3 triggers with the same identifier get stored in the same server and in this solution the single server has to forward packets to all the receivers. This makes the server a bottleneck and it is not scalable. To scale this solution the fact that triggers can be chained is used. Servers put a limit on the number of triggers that they accept for an identifier and once this number is exceeded a new node inserts a trigger to another identifier and also a trigger to forward messages to the server responsible for this new identifier. This essentially creates a delivery tree. We do not discuss the detailed algorithm here.

Overcast

Overcast [47] is an application level multicast implementation that targets applications with high bandwidth requirements. Overcast is a single source reliable delivery multicast protocol. A new node joins the overcast network by contacting the root node. Then it tries to move away from the root as much as possible on the tree without sacrificing the bandwidth. This is done by measuring the bandwidth to the root through the current parent and also through the siblings. If it gets good bandwidth through a sibling then the node becomes a child of the sibling. The new node reevaluates this decision periodically by comparing the bandwidth through the current parent with the bandwidth through its former parent.

Overcast makes a distinction between the nodes in the delivery tree and the clients. The clients connect only to receive data and they do not participate in data forwarding. The root node keeps complete information about the nodes in the delivery tree so that it can direct the clients to a suitable node to join and receive data. To keep this information current, overcast also has a protocol that exchanges death and birth certificates of nodes and other information about the nodes.

NICE

In contrast to Overcast, the application level multicast protocol, NICE, presented by Banerjee et al. [7] is for low bandwidth data streaming applications. The NICE protocol also assumes a small set of receivers. NICE builds a mesh based control topology and does not build explicit delivery trees. The data delivery tree is implicit in the control topology. NICE assigns members to a hierarchical set of layers and each layer is partitioned in to several clusters of sizes between k and $3k - 1$. All hosts are members of the lowest layer (layer 0) and the leaders of the clusters in layer 0 are members in layer 1 and so on. There is only a single host at the highest layer.

Soft control state is exchanged between the peers in each cluster to maintain certain invariants (such as the maximum size of the cluster). Note that a host can belong to clusters in several layers. The sender specific implicit forwarding tree is built as follows. The sender sends the message to all the members of its cluster in level 0. This message is received at the cluster leader who in turn sends it to the members of its level 1 cluster. This message is also received at the leader of this cluster who are a member of the level 2 cluster and so on. Cluster leaders of level 0 clusters other than the cluster of the sender also forward the message to the members in their level 0 cluster. This procedure is repeated at all levels and this ensures that messages are delivered loop free.

End-system multicast (Narada)

Chu et al. [27] present the *Narada* protocol for end system multicast. Narada targets applications with medium sized groups. Narada creates a mesh topology connecting participating hosts and builds source specific delivery trees on top of this mesh. Narada requires that each member of this mesh maintain a list of all the other members. The authors note that since Narada is aimed at medium sized groups this requirement is not a heavy burden. To make sure that the topology is not partitioned each member sends refresh messages with monotonically increasing sequence numbers to all the other members through the mesh. To make this process scalable, instead of flooding the messages along the mesh, each mem-

ber periodically exchanges its knowledge of members with its neighbors. To join the mesh a new member must somehow know at least one member already in the mesh. Note that the mesh topology is not a fully connected topology. Narada has a mechanism to add and remove links to the mesh based on the utility of each link. Narada uses a routing protocol similar to DVMRP to construct source specific delivery trees on top of the mesh.

The Narada implementation has been used to multicast several live events including the 2002 ACM SIGCOMM conference.

Bayeux

Bayeux [99] is a source-specific multicast system built on Tapestry [98]. It builds a multicast tree rooted at the source. In Bayeux each multicast session has a unique identifier. To create a multicast session a one way hash function is used to map this identifier to a 160 bit identifier. Then a "file" named with that identifier is placed in the root node of the session. The original session identifier is advertised over Tapestry.

A node that wishes to join a session also uses the same hash to generate the 160 bit identifier and uses this identifier to locate the "file". This way the new nodes contact the root by sending a JOIN message and the server replies with a TREE message. The TREE message causes each router along the path to insert an entry in its forwarding table for that session. This forms a delivery tree.

Scribe

Scribe [22], built on Pastry [84], provides best effort multicast without any particular delivery order. Any source can send messages to the group but there is a single shared delivery tree.

To create a group a node routes a CREATE message over Pastry with the group identifier as the key. Pastry delivers this message to the node with an identifier numerically closest to the group identifier. This node acts as the root of the tree and also as the rendezvous point. When a new node wants to join it sends a JOIN message to the rendezvous

point over Pastry. When this message reaches a node along the path it checks whether it is a forwarder for the group and if not it becomes a forwarder and sends a JOIN message towards the root. The original message is terminated. If it is already a forwarder it simply adds the new node to its children table.

Senders send multicast messages via the rendezvous point. Senders first locate the rendezvous point through Pastry and then cache its IP so that messages can be sent directly to the rendezvous point.

2.1.3 Reliable multicast in group communication systems

Multicast (or broadcast) provided in group communication systems are different from the multicast schemes that we described so far in providing strict delivery and ordering guarantees. These multicast systems also show the diverse possibilities in communication paradigms as well as different implementation strategies.

Birman et al. [16] present several communication primitives to send reliable messages to a group of processes. ABCAST and CBCAST are two of them. These primitives are termed as broadcast protocols rather than multicast. ABCAST is an atomic broadcast that guarantees that all the operational processes receive a message or none at all. Two messages from the same process are delivered in the same order as they were sent. CBCAST messages are causally ordered. This is weaker than the ABCAST primitive in the sense that only causally related messages are ordered. The ISIS [15] toolkit provides implementations of these broadcast primitives. The Spread [2] system implements the same primitives over a wide area network. The Ensemble [14] system provides a framework to compose different group communication protocols by stacking a set of micro-protocols. This system also allows different implementations of the protocols to be used transparently to the application.

2.2 Gather

We discuss several related communication paradigms under the heading of gather. These are the communication paradigms that are sometimes referred to as the inverse of the multicast, where multiple sources send messages to a single receiver.

2.2.1 Concast

Calvert et al. [19] present *concast* as a network service and identify it as the inverse of multicast. Concast provides a scalable service to applications that require a receiver to collect messages from many senders. In contrast to gathercast [4], which is an optimization method in the network transparent to the applications, concast is proposed as a service provided by the network. The participants explicitly join and leave a concast session.

In concast, the senders are identified by a single group address and the single receiver does not have to be aware of the identity of the senders. This is similar to multicast where receivers are hidden by a group address. The datagrams sent by the senders contain the group address and the unicast address of the receiver.

Two varieties of concast, *simple concast* and *custom concast*, are described in [19]. The simple concast suppresses "identical" datagrams by different senders and only delivers one such datagram to the receiver. The authors describe a mechanism in which the receiver uses the modified IGMP to indicate the groups that it wants to receive from. The network has to maintain the state for each flow, identified by the group address and source address pair, to suppress the duplicate packets. In custom concast the receiver can insert a merge specification into the network to merge the packets. Both varieties need the support of the network and Calvert et al. [19] suggest that custom concast can be implemented in Active Networks.

2.2.2 Gathercast

Gathercast [4] is a multipoint-to-point aggregation mechanism. Badrinath et al. [4] discuss this as a mechanism implemented in the network to combine smaller packets into larger packets transparently to the senders and the receivers. The authors note that there are applications that generate a large number of small messages that must be aggregated at a single point.

Gathercast combines smaller packets to the same destination and sends fewer packets over the network. This needs the support of the routers. The receivers are allowed to place aggregators in the network. This needs active network functionality from the network.

2.2.3 Collect primitive in Macedon

Macedon [82] is a framework to build application level overlay networks. Macedon provides a generic API that these networks export. One of the API calls provided in Macedon is `macedon_collect()`, which collects messages from non-root nodes in a delivery tree at the root of the tree. The messages travel towards the root over the tree and the authors mention that the nodes along the tree can also summarize data in an application specific way. Macedon only provides the API to be exported by the overlay. It is up to the overlay built using Macedon to provide the exact semantics for the API.

2.3 Anycast

Anycast is the communication paradigm where a message is sent to any of the receivers. The senders are not concerned with the identity of the receivers.

2.3.1 Application-layer anycasting

Bhattacharjee et al. [12] emphasize on distinguishing anycast as a communication paradigm from its implementations. They discuss an application layer implementation of anycast based on *anycast resolvers* which are similar to DNS name resolvers. Anycast resolvers

resolve Anycast Domain Names (ADN) to IP addresses. An ADN identifies an *anycast group* which is a collection of either unicast or multicast IP addresses. The anycast service resolves the ADN to one of the IP addresses according to some criteria. When the anycast group consists of multicast addresses this is somewhat similar to manycast, even though the authors do not explicitly mention this.

Among the applications that can benefit from their service, the authors mention that some quorum consensus protocols can use the anycast groups with multicast addresses—each subset of the set of coordinators is assigned a multicast address and included in the anycast group and the service maps a query to the set with the least loaded set of coordinators.

2.3.2 IP anycast

Although there has been an informational RFC [73] describing anycast for IPv4, it has never been implemented. However, anycast-like services have been in use in the form of content distribution networks such as Akami which directs clients to a nearest server. Metz [67] mentions multiple servers in a LAN hidden behind a single virtual IP address and DNS servers that map a name to multiple IP addresses in round robin fashion as forms of anycasting. Metz [67] also discusses possible ways to deploy anycast in IP networks.

Katabi et al. [51] present an anycast architecture (called GIA) for the Internet that has its own anycast address range. Katabi et al. note that anycast groups represent services. They also note that not all services are popular at a given site. Rather than trying to provide optimum routing for all anycast addresses, in GIA routers provide efficient routes to popular anycast addresses. In GIA an IP anycast address has a variable length prefix denoting its "home domain". It is the unicast prefix of the domain of the owner of the anycast address and there must be at least one member in that domain. This home domain provides a default destination and for unpopular groups messages are delivered to the network of the home domain. If this packet goes through a routing domain that has information on another nearest group member the packet can be delivered there. Efficient routes to popular groups

are searched using modified BGP.

Anycast [48] proposed for IPv6 [31] only allows anycast addresses on router interfaces and not on host interfaces. IPv6 anycast addresses are allocated from the unicast address space. All the members of an anycast group must be within a region that shares an address prefix. Anycast addresses are not advertised separately outside this region. Within the region anycast routes are advertised individually.

Anycast using i^3

Lakshminarayanan et al. [57] also discuss a mechanism to implement anycast on i^3. All anycast receivers in a group insert triggers for identifiers with the same k most significant bits. The senders send messages to an identifier with the same most significant k bits. The Pastry routing would send a message to the server that is responsible for the longest prefix matched identifier and that server forwards it to the receiver that inserted the trigger. This ensures that a message is received at only one receiver.

2.3.3 Anycast on Scribe

Castro et al. [23] present an implementation of anycast on Pastry [84]. This implementation uses the multicast delivery tree built by Scribe (we discussed Scribe in Section 2.1.2) for anycast. An anycast message is routed towards the root of the group tree over Pastry. If a node on the path is already on the tree, it starts a depth first search from that point in the tree to find a member of the group; note that not all the nodes on the delivery tree are members of the group. If a member is found then the message is delivered to that member. If the search could not find any member then the node that initiated the search checks whether it is a member. If that is the case then the message is delivered to itself and if not, the message continues towards the root and the message also carries the node identifier of that node as a visited node. Once all the nodes have been visited, finally, the message is delivered to the root if it is a member. If the root too is not a member then the group is empty and an error is returned to the sender of the message. Because of the locality awareness of Pastry

the members of the group in the subtree rooted at a tree node closest to the sender of the message are the closest members. This ensures that an anycast message is delivered to a member of the group that is close to the sender.

2.3.4 Anycast in generic overlay APIs

The generic API presented by Dabek et al. [30] for structured peer-to-peer overlays includes anycast. Macedon also includes anycast in its generic API for overlay networks.

2.4 Manycast

Manycast delivers a message to a sender specified number of members in a group. Some [21] have defined manycast also as one-to-many-to-one communication where the recipients also reply back.

2.4.1 Manycast on Scribe

The anycast implementation on Pastry [23] also supports manycast. Manycast is achieved using the same algorithm as above but by iterating it until the message is delivered to the required number of group members. The message carries the number of recipients that have already received the message.

2.4.2 Manycast in mobile ad hoc networks

Carter et al.[21] propose manycast in ad hoc networks in the network layer. They identify manycast as a bidirectional one-to-many-to-one communication. A manycast packet is delivered to k out of m receivers but there is no delivery guarantee. Carter et al. propose an API that lets clients specify k, the number of responses expected, and the reliability level. They present several routing algorithms (flooding, scoped flooding, small group multicast etc.) to implement manycast and also present comparison of these using simulations.

They conclude that manycast is viable in ad hoc networks and must be implemented at the network layer.

2.4.3 PAMcast

PAMcast [24] is a generalized manycast service (authors define it as a generalization of anycast and multicast). PAMcast delivers a message to some specified number of group members. The PAMcast service selects the set of members (of the size specified) according to the *mode* of the request. In the *balanced mode* the message delivery set is selected to balance the load and in the *closest mode* the recipients that are closest to the root of the delivery tree are selected (PAMcast uses a shared tree). The authors describe an architecture of the PAMcast service that can be implemented on the routers in the network or as an application level overlay network. The implementation discussed in [24] provides best effort delivery.

To build and maintain the delivery tree PAMcast uses PAMcast Group Membership service (PGMP) which is somewhat similar to IGMP. In the router based implementation when a new node joins, a message is sent to a router that is already in the delivery tree and also that message is propagated all the way to the root of the delivery tree. On the way up the tree this join message updates the degree of the tree rooted at each router all the way to the root of the delivery tree. When an application PAMcasts a message, the message is sent to the root of the delivery tree and root checks the number of required recipients (degree) for this message. Then it checks the degree of its subtrees and a copy of the message is sent along the selected sub trees with modified message degree so that the total message degrees of the duplicated messages is equal to the degree of the original message. It also guarantees that each sub tree degree is larger than or equal to the degree of the message sent along that tree.

Quorumcast

Cheung et al. [25] present quorumcast as sending a message to any arbitrary q-subset (quorum group) of m nodes. Despite its name this is similar to manycast. Once the quorum group is selected a routing tree is constructed to deliver messages to the group. Cheung et al. note that selecting q randomly results in inefficient delivery trees. They present several heuristics to find a minimum cost quorumcast routing tree. These heuristics assume that the source node has all the information about the network.

2.5 Other communication paradigms

2.5.1 Somecast

Somecast [96] is a communication paradigm of receiving from some number of senders out of a group of senders. This very interesting communication paradigm is presented with respect to a single application, but a generalization is possible. Yoon et al. [96] present somecast as a receiver receiving from some of the several concurrent multicast sessions according to its Quality of Service (QOS) requirements. The receiver joins as many multicast sessions as required to satisfy the QOS requirements. In this particular application the data multicasted by the senders are encoded using a form of Reed-Solomon code so that data blocks from multiple senders can be combined to obtain the original object.

Hefeeda et al. [42] present collectcast which also has a multiple sender and one receiver communication pattern and it is also receiver initiated communication. Collectcast is also presented in a very application specific setting.

2.5.2 Probabilistic multicast

Ammar [3] discuss probabilistic multicast as the multicast-to-some generalization of the traditional multicast. In probabilistic multicast messages are sent to a multicast group, but each receiver has an *acceptance probability*. Ammar discuss three methods to determine the acceptance probability and in one method the probability is carried in the messages sent.

In this form the probabilistic multicast is quite similar to manycast. Instead of denoting a number of receivers, the sender denotes a probability of acceptance. In another method the receiver itself can define the acceptance probability of a message which makes it quite different from manycast. Ammar also describes a third method of defining the acceptance probability in which the probability is determined by combining the probability carried in a message and the probability assigned by the receiver.

2.6 Message passing interface (MPI)

MPI [65, 66] provides a set of collective communication primitives to use in message passing parallel programs. MPI only defines the interface of the communication primitives and does not describe an implementation for them. The collective communication routines improve the expressiveness of the application. Rather than composing all the communication using point-to-point communication primitives application programmers can chose a collective primitive that best expresses the communication patterns of the application.

This clean interface allows applications to use the communication primitives (not just the collective communication primitives) independent of their implementations. The standardized API has spurred innovative implementations and immediately after the release of the standard several implementation were available such as LAM/MPI [58, 18] and MPICH [40]. These implementations explore the design space independent of the applications.

The design space in implementing communication primitives, especially the collective communication primitives, is vast. Squyres [85] notes this vast design space and comments that one set of collective algorithms is not sufficient for different applications and runtime systems. LAM/MPI has recently been redesigned to allow different implementations of communication primitives to be plugged in easily [85]. The standard MPI interface allows progress in these algorithms independent of the applications.

While these features coincide with our objectives, the MPI model is not the right model for the applications that we have in mind. As the name suggests, the MPI collective

communication routines are truly collective; all the nodes in the group (communicator [65]) must call the routines in unison. MPI is well suited for message passing parallel applications, which are designed to perform a cohesive task and are distributed to achieve good performance. However, such a communication model is not common to all distributed applications.

Another restriction of the MPI communication model comes from the static nature of the communicators (even in MPI 2.0 [66] communicators are static). Any addition or deletion of a process can only be done by creating a new communicator. This again needs agreement from all the processes in the current communicator through a collective call. This is vastly different from the communication paradigms that we discussed above. The communication paradigms we discussed so far provide a level of indirection between the participants of an instance of that communication paradigm. For example, in IP multicast the participants—senders and receivers—interact through a group address. The group can grow and shrink without informing the members of the group. Rather than separating the senders and the receivers MPI requires senders and receivers to rendezvous; even in the case of ANY_SOURCE receiving there must be matching pairs. Furthermore, a failure of a process in an MPI communicator leaves that communicator in an invalid state, resulting in a potential failure of the entire application [35]. This defeats the goal of many distributed applications, which are distributed to avoid a single point of failure.

The set of communication primitives is fixed in MPI; this has to be since it is standardized. While this standard API spurs innovation in implementing the primitives, it also inhibits the innovation along the line of communication primitives. As we have seen in the communication paradigms presented here, there is a possibility for novel communication paradigms. A standard cannot anticipate all these paradigms and provide APIs for them.

There is such a difference between MPI and the above mentioned communication paradigms because the target domain of the applications are different. The target domain of MPI is the domain of parallel applications. The requirements of parallel applications widely differ from the requirements of distributed applications. Although MPI provides

motivation for this work, we recognize that the communication requirements of distributed applications are different from those of parallel applications.

Chapter 3

Issues

In Chapter 2 we discussed several multiparty communication paradigms and several implementations of them. We make two key observations of the material presented in Chapter 2.

1. There are large number of communication paradigms. We presented the multicast, anycast, manycast, gather (concast, gathercast), and somecast (receiving from some number of senders) paradigms. The space of communication paradigms is larger than evident from the above terms. That is because the terms such as multicast stand for families of communication paradigms rather than a single well defined communication paradigm. For example, multicast is used to identify *single-source-multicast* as well as *all-to-all-multicast*. Express multicast is an example of the former and IP multicast is an example of the later. Such a fine distinction of communication paradigms is useful for two reasons; users can clearly identify the communication paradigm that they want to use and implementors can take advantage of the narrowed specification to provide optimization. This second point is evident from the scalability of Express multicast as opposed to IP multicast.

2. There is a vast design space to be explored in implementing these communication paradigms over the Internet. The large number of multicast implementations and algorithms are the evidence of this design space. Implementing any one of these communication paradigms over the Internet is not an easy task. This task becomes

even harder when one tries to optimize the performance along different metrics.

These communication paradigms provide a powerful set of abstractions to distributed-application programmers. Ideally, application programers should be able to use these communication abstractions in their applications independent of their implementations. This allows application programers to concentrate on their programs while allowing the implementors of the communication paradigms to explore the design space. This also provides an opportunity to modularize the communication. This idea is reminiscent of the collective communication primitives in MPI [65, 66] and techniques such as Script [37] for concurrent programming languages that allow modularizing communication patterns.

We identify three unresolved issues that hinder the use of communication paradigms in applications independent of their implementations.

1. Naming.
2. Application programmer's interface (API).
3. Support structure to implement and deploy.

We discuss these issues in detail in the following sections.

3.1 Naming

We made the observation that there are large number of communication paradigms. If applications are to use these communication paradigms, application programmers should be able to identify them precisely. Also the implementors of these communication paradigms should be able to uniquely identify the paradigms they implement. This is essential to separate the use of the communication paradigms from their implementations. Therefore, we should be able to name[1] them uniquely. However, currently there is no common name space to identify all the communication paradigms we presented in Chapter 2.

[1]For convenience we use the term *name* to refer to an identifier of a communication paradigm. We do not use it to refer to a particular form of that identifier; it can be a human readable string, a specification or some other form.

We observe that the ad hoc names used to identify multiparty communication paradigms have the following problems.

1. Several related communication paradigms are identified by the same name.
2. The same name is used to identify completely different communication paradigms.
3. Different names are used to identify similar communication paradigms.
4. The use of confusing names.

A good example of the first problem is the name multicast. We showed earlier that multicast in fact stands for a family of communication paradigms. It is important that we develop individual names to identify these communication paradigms lumped together under the term multicast. For example, an application that requires the all-to-all multicast paradigm cannot use an implementation that implements single source multicast. The distinction between these two communication paradigms is important for implementors too; implementors can take advantage of the fact that there is guaranteed to be only a single sender in single source multicast to optimize and simplify the implementation. Express uses this fact to implement multicast in a scalable manner as opposed to IP multicast which is in fact all-to-all multicast. Birman et al. [14] discuss the use of FIFO ordered multicast instead of the total ordered multicast as long as there is only a single sender, in the Ensemble system. The system dynamically switches to total order multicast when a second sender starts sending. If the application is given a chance to identify the exact communication paradigm to be used, then the implementation could have used the optimized version directly.

This naming issue is not only confined to multicast. For example we can identify several communication paradigms lumped together under the term anycast. For example, it is not clear in anycast whether there is exactly one receiver of the message or one or more receivers of the anycasted message. It is also not clear whether the receiver of the message replies back to the sender or not. IPv6 anycast has different semantics than the anycast implementation presented by Castro et al. [23].

An example of the second problem is manycast. Yoon et al. [96] use the term manycast to identify the communication pattern of receiving from multiple senders. In contrast, Castro et al. [23] use the same term to identify sending a message to any n number of receivers—a generalized anycast paradigm. Carter et al. [20] define manycast as a single client sending a request to many servers and all of them answering back. This use of the term manycast to mean completely different communication paradigms highlights the problem of using arbitrary names.

Calvert et al. [19] use the term concast to refer to a communication paradigm described as inverse multicast. Macedon [82] provides a primitive, *macedon_collect()* which is introduced similarly as the opposite of multicast. Both allow for summarization of data in an application specific manner. Despite their similarity they use completely different names. This is an example of the third problem.

An example of the fourth problem is somecast. Somecast gives an impression that it refers to sending to some of the receivers. However, that name has been used to refer to a communication paradigm in which the receiver is receiving from some of the senders [96]. This shows the folly of using everyday terms to describe the communication paradigms.

Not all names used to identify communication paradigms have these problems. In the domain of group communication there are well defined and uniquely named communication paradigms. For example, several variations of multicast with respect to delivery guarantees are discussed by Birman et. al [16]. They use terms such as ABCAST and CBCAST to identify the different variations. These names uniquely identify communication paradigms and we have not seen these names being used to identify communication paradigms other than the intended ones. However, these names do not come from a name space that accommodates other communication paradigms and not even other multicast-like communication paradigms.

Languages such as CSP and LOTOS have been used for specifications of reliable multicast protocols [29, 1]. However, there are many protocols that can be used to implement any given communication paradigm and a particular protocol does not identify the

abstract idea of that communication paradigm. Protocol specification languages lead to specifications in terms of set of actions and hence a particular implementation. We revisit this issue in Chapter 4 where we can discuss it further in a more appropriate setting.

MPI has a large collection of multiparty communication paradigms known as collective communication primitives. MPI addresses the naming issue by having a fixed set of communication primitives with a standard set of names which are also tied to the API. These names by themselves do not convey full details of the communication primitives, but the standard rigorously defines the semantics of each communication primitive. MPI standardization was a huge process that involved about 60 people from 40 different organizations [65]. Such a standaridzation process is not appropriate for most of the communication paradigms we discussed in Chapter 2. Standardization is especially not desirable considering the fact that we do not know the extent of the space of communication paradigms.

Bayerdorffer [9, 8] identifies that the specification of patterns of communication as a main complexity in concurrent programming. He mentions the importance of naming these communication patterns. He presents a concurrent system named *Associative Broadcast* that allows programmers to specify complex patterns of communication through its naming convention. Associative broadcast names the target set of receivers using first order logic and that specification is carried with the messages and messages are broadcasted. Messages are delivered only to the recipients with properties that satisfy the specification. Such a system is not suitable for our purpose for three reasons; 1) Associative broadcast does not name the communication pattern directly. The communication patterns are expressed indirectly by defining the specification of the intended recipients. 2) Only one-to-many $(1 - N)$ communication patterns can be described. 3) The associative broadcast system and the naming system is defined with the assumption of a broadcasting medium such as the Ethernet. In contrast, one of our motivations for this work is the lack of such an environment over the wide area Internet.

One can envision a naming scheme that names a communication paradigm by placing it in a space defined by several properties expected of the communication paradigms.

For example, such a property list may include guaranteed delivery, ordered delivery, single source, etc. And the communication paradigm may be named by placing it in a point in the space defined by those properties. However, this also limits the name space to one that can be defined by the known properties. One lesson that we take from the diversity of the communication paradigms presented in Chapter 2 is that it is not prudent to limit the naming space only to accommodate the known communication paradigms.

We expect that the set of multiparty communication paradigms are larger than the set that has been discussed in the literature. In such a scenario we need a name space that can accommodate all these communication paradigms and name them precisely. An important feature we expect from an ideal naming scheme is that it should anticipate new communication paradigms as well as existing ones. To allow for innovation and rapid deployment the name space should not be a centrally administered one either.

3.2 Application programmer's interface (API)

The papers that presented multicast implementations ALMI [75], End-System multicast [27], and Overcast [47] do not mention an application programmer's interface at all. Liebeherr et al. [60] note that these implementations do not provide an explicit general purpose API and also note that they integrate the application program and the software responsible for maintaining the overlays. It is understandable that these implementations do not have a compelling reason to consider an API as an important component. These implementations present interesting tree construction algorithms and message routing protocols to optimize certain metrics. They are not addressing the problem of using multiple communication paradigms or multiple implementations of a communication paradigms in an application. These efforts are concentrated on exploring the design space of a single communication paradigm. Therefore, a well defined API is not a compelling requirement.

We emphasized that there is a large design space in implementing any given communication paradigm. If the implementors of a communication paradigm are to explore the design space independent of the applications that use it, it is important that all the imple-

mentations provide a common API. Without such a common API an application cannot use a different implementation of a communication paradigm easily without substantial changes to the application.

Several researchers have identified the need for a common API for overlay networks [60, 30, 82]. Liebeherr et al. [60] present the *Overlay Socket* as a common API for programming overlay networks. An overlay socket hides the topology of the overlay network and application programmers use a common API irrespective of the overlay topology. The API provides a `sendToAll()` method to multicast a message and a `sendToNode()` method to unicast a message. The interface also provides methods to send messages to nodes identified as parents (`sendToParent()`), children (`sendToChildren()`), and neighbors (`sendToNeghbors()`). The overlay socket API exposes the overlay network to the programmers; after all it is an API to program overlays. There are only two communication paradigm specific methods, `sendToNode()` for unicast and `sendToAll()` for multicast. Overlay sockets clearly do not address the issue of providing APIs for other communication paradigms.

Dabek et al. [30] provide a common API for structured peer-to-peer overlay networks. They provide a common *key based routing* (KBR) API at the lower level and also identify several categories of abstractions as higher level abstractions. These are distributed hash table (DHT), anycast and multicast (CAST), and decentralized object location and routing (DOLR). In the CAST abstraction they identify `join()`, `leave()`, `anycast()`, and `multicast()` as the API calls. They also mention that defining an interface for the CAST, DOLR, and DHT abstractions is ongoing work. At the KBR level the API includes calls to route and deliver messages, an upcall to forward messages, and an interface to access the routing state of the node. While the KBR interface exposes the overlay to programmers, the CAST interface provides APIs for two communication paradigms anycast, and multicast. However, these are the only multiparty communication paradigms provided through the generic API.

Macedon [82] provides a generic API for overlays as well as a language to build

overlay networks. The authors mention that they provide an API similar to the work by Dabek et al. [30] mentioned above. Apart from anycast and multicast interfaces Macedon also provides the `macdeon_collect()` interface to collect messages from non-root nodes to the root over the distribution tree.

In all the three works mentioned above the APIs do not mention the exact semantics of multicast, anycast, and collect. They leave the exact semantics to the implementations. This diminishes their value as generic APIs. These three APIs also highlight the fact that there is no general agreement on what multiparty communication paradigms (even with imprecise definitions) are to be provided—overlay sockets provide only multicast, the API for structured peer-to-peer networks provides multicast and anycast, and Macedon provides multicast, anycast, and collect. The fact that these communication paradigms are included in these APIs argues for their importance. The progressive inclusion of communication paradigms also shows that we do not know all the potential and useful communication paradigms.

A good example of a well defined API is MPI. The MPI standard defines the exact semantics of each API function. In fact, MPI addresses both the issues of naming and the interface in one stroke by standardizing the communication primitives and the interface. As we discussed before standardization essentially limits the number of communication primitives and it is not suitable for wide variety of communication paradigms we observe in the distributed computing world. Standardization is also a barrier to invent communication paradigms.

An ad hoc API for every implementation of a communication paradigm makes it harder to use new implementations of a given communication paradigm in applications and a standard API for each and every communication paradigm (as in MPI) makes it harder to innovate new communication paradigms. A common API that is generic enough to accommodate a large number of communication paradigms allows for innovation without hindering the use of different implementations of a given paradigm in applications. Such an API should not be an arbitrary API or an API that is based only on the known set of

communication paradigms. Ideally, we need an API that can accommodate all the communication paradigms that can be named under a precise naming scheme for multiparty communication paradigms. This is an ambitious goal considering the wide variety of requirements of multiparty communication paradigms. In multicast the processes require a mechanism to send and receive messages. Same for anycast and manycast. In all these cases a mechanism to join an existing instance of the communication paradigm and also a mechanism to instantiate such an instance are essential. Manycast must provide an interface to set the number of recipients that should receive the message. The same requirement is there for somecast-like paradigms, where the receiver decides on the number of senders that it wants to receive from. In a communication paradigm that allows processes to select the role (sender or receiver) dynamically, it is useful to have an interface to select these roles. For example, consider a hypothetical anycast communication paradigm where the messages are directed towards pre-declared receivers. In such a case an interface that allows this role to be set is useful.

The effectiveness of any API has to factor in how programmers perceive that API since ultimately it is intended to be used by application programmers. A generic API for all communication paradigms may not be the preferred API for all programers. Also it may not be amenable to different styles of programming. For example, an API that only provides blocking calls may not be easy to use in a program written in event driven programming style. Furthermore, any given API may not have straight forward bindings to all programming languages. A case in point is the problems binding MPI to Fortran.

> "As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. The violations were originally adopted and have been retained because they are important for the usability of MPI. . . ." [66]

These are problems associated with any API and we do not address these issues in this

work; we address the issue of deriving a common API for communication paradigms.

A common API itself does not provide all the information to correctly use that API for a given communication paradigm. For example, only a single process in a single-source multicast session is allowed to send messages and all the other processes can only receive messages. Given a communication paradigm we also need a mechanism to identify how the common API can be used for the communication over that communication paradigm.

3.3 Support structure to implement and deploy

The multiparty communication paradigms we described in Chapter 2 and their implementations can be considered as *disruptive technologies*. Christensen [26] uses the term *disruptive technologies* to denote technologies that are innovative and that introduce new value propositions to the market; they may not even perform well according to the currently accepted metrics, but may eventually even replace the current technologies. Such technologies may not get mainstream acceptance to be deployed globally. A report by the Computer Science and Telecommunication Board (CSTB) of the US National Research Council [28] discusses the problem of deploying disruptive networking technologies. They note that the eventual acceptance of disruptive technologies are tied to applications that built on top of these technologies. However, for applications to use them the new technologies must have a wide deployment.

> "...It is hard to know up front what the "killer app" for new enabling technologies will be, and there are no straightforward mechanisms to identify and develop them. With any proposed technology innovation, one must gamble that it will be compelling enough to attract a community of early adopters; otherwise it will probably not succeed in the long run. This chicken-and-egg-type problem proved a significant challenge in the Active Networks program (as did failure to build a sufficiently large initial user community from which a killer application could arise)." [28]

It is not reasonable to expect even a few of the multiparty communication primitives to be available in the global Internet. Lack of general deployment of IP multicast in the Internet more than 15 years after its acceptance into the IPv4 protocol suite is a classic example of the near impossibility of deploying new technologies over the Internet. This ossification of the Internet is discussed in the CSTB report [28] and it suggests the overlay approach as a viable method to deploy disruptive technologies. Some of the implementations of communication paradigms have been implemented as overlays (DHT or otherwise).

Simply providing an overlay based solution does not completely solve the deployment problem. Take for example deploying a DHT based multicast scheme such as Scribe. Scribe trees are constructed on a Pastry DHT. If only a few processes want to use Scribe based multicast there is no point in building a DHT. Only a public DHT could solve such a problem. There is an effort to deploy a public DHT [72] over the PlanetLab [77, 76], which itself sprung as an overlay based solution to deploy disruptive technologies, as a solution to such problems.

PlanetLab is only limited to 400+ nodes and its use is largely limited to the research community that takes part in its deployment. The CSTB report [28] also note that the "killer apps" that resulted in the popularity of the Internet and the PCs did not come from the research community. Therefore, it is essential that the communication paradigms be deployed in such a way that general users can use them to develop applications.

Any infrastructure-based deployment effort, such as Mbone for multicast deployment, requires a huge effort and coordination. Non infrastructure-based truly end-system overlays have more flexibility to be deployed where and when users want them. In such a model users could use an implementation of a communication paradigm by simply linking to a library. This is the current method of deployment adopted by most implementations of communication paradigms.

There is a possibility that the applications that communicate over a single session of a communication paradigm be developed by different application programmers. For example there are Instant Messaging Clients developed by different programmers to be

run on different platforms. It is also possible that different sessions that these applications participate in use different implementations of a communication paradigm. For example one multicast[2] session may use the End-System multicast implementation and another may use ALMI. In such a situation we need a mechanism for all these applications to agree on the exact protocol (such as Narada) that implements the communication paradigm. Therefore, a deployment solution that simply expects applications to link to a library that provides an implementation is not suitable.

Deployment spurs innovative applications. We also expect innovations in communication paradigms and their implementations. This requires support to implement them. The multicast implementations such as End-System multicast, ALMI, and Overcast all build overlays. They all duplicate the functionality common to all overlays, such as message forwarding and measuring overlay links. The time and effort required to duplicate similar functionality can be avoided by providing these functionalities as a library or in the run-time system. Ideally, the exact method of monitoring the overlay links and also creating these links should be decided at the deployment time to suit the deployed environment. This allows communication paradigms to be implemented by concentrating the effort on innovative algorithms rather than worrying about how to extract information specific to the deployed environment.

3.4 Solutions

In Chapter 4 an abstract model is presented to address the first two issues; naming and API. We present the notion of *multiparty communication types*, which are the precisely defined and named counterparts of the multiparty communication paradigms. In Chapter 5 an architecture of a middleware system, MayaJala, based on the abstract model is presented. MayaJala provides a supporting system to build, deploy, and use communication types thus addressing the third issue.

[2]We do not use the term multicast here to indicate any precisely defined communication paradigm. We use it in its loose sense.

Chapter 4

A Model for Multiparty Communication

We develop an abstract model of multiparty communication that addresses two of the issues raised in Chapter 3; naming and application programmer's interface (API). In Chapter 5 we use the derived API and the concepts developed with the abstract model to design and implement a middleware system.

The abstract model describes *multiparty communication types*. Multiparty communication types are the precisely defined and named counterparts of the loosely defined and named communication paradigms. The multiparty communication type system gives a naming scheme to address the naming issue described in Chapter 3. From this precise naming scheme we derive a simple API. By naming a communication type using this model we also identify its API.

The model we present is a very simple one intended to show the validity of the idea of communication types. Our contributions are the idea of communication types and the other concepts associated with it and we use the model as a vehicle to present these ideas.

Note that the communication types are not intended to be refined into implementations or act as specifications that can be used to validate implementations. Communication types simply provide a classification system for communication paradigms.

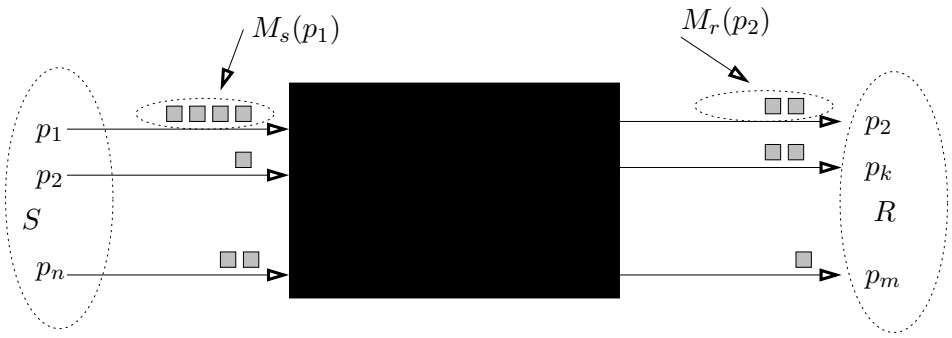

Figure 4.1: The black box model of multiparty communication.

4.1 Multiparty communication types

To define multiparty communication types we use a model that consists of a black box and a set of processes, P, connected to that black box (Figure 4.1). Each process p can be *send enabled* in which case $p \in S$. Similarly the *receive enabled* processes are in the set R. A process p can be in both S and R. The processes send messages to the black box and the black box delivers some combination of these messages to the processes. The black box only delivers messages to processes in R and only processes in S can send messages into the black box.

Let Σ be the alphabet of message identifiers. We define three functions M_s, M_r, and N as follows.

$$M_s : \ P \to \Sigma^*.$$

$$M_r : \ P \to \Sigma^*.$$

$$N : \ P \to \mathbb{N}.$$

There is a value $N(p)$ associated with each process $p \in P$. The messages sent into the black box by a process $p \in P$ are modeled by the sequence $M_s(p)$ and similarly messages received are modeled by the sequence $M_r(p)$. Each message in $M_s(p)$ is globally unique. All received messages must have been sent. That is,

$$\forall_{q \in P} \, \forall_{m \in M_r(q)} \, \exists_{p \in P} \, m \in M_s(p). \tag{4.1}$$

Note that we use the symbol $\in$ to indicate the membership of a set as well as to indicate the membership of a sequence. The exact meaning is clear from the context.

Once a process is in P it cannot leave P. The size of P monotonically increases. The sets S and R can grow and shrink. When joining the set P a process p must decide on its membership of S and R and the value of $N(p)$. If a process p does not specify its value when joining we consider $N(p) = 1$. A process p changing its membership in the sets S or R or changing its value is modeled as p simply leaving both S and R and re-joining P as a new process.

We model the state of the system that consists of the black box and the set of processes P by the tuple:

$$\delta =< P, S, R, N, M_s, M_r >$$

We call such a tuple a snapshot. A change to P, N, S, R, M_s, or M_r results in a new snapshot. A sequence of such snapshots, Δ, describes the operation of the system. Such changes are governed by the following rules, in addition to the rule 4.1.

- When a process sends a message, M_s changes and this results in a snapshot.
- When the black box delivers a message to a process, M_r changes and this results in a snapshot.
- When a process joins P it simultaneously joins S and R (or one of them). This results in a single snapshot. This simultaneous inclusion into S and R does not deviate from the reality. In practice, the choice of being send enabled or receive enabled is a local decision of a process and no global synchronization is necessary. The sets S and R are simply abstract entities to help us reason about the global state. Note that only a single process can be added in a single snapshot.
- When a process leaves S and R, that results in a single snapshot. Note that when a process leaves S or R it is still in P and hence P does not change. Once a process leaves S and R it will never be allowed back into S or R.

Note that the processes are asynchronous and hence there are many different possible sequences of snapshots. For example, messages sent by two processes results in two snapshots which may be ordered differently in two possible snapshot sequences.

The black box expects S, R, N, and P to have a certain property in all possible snapshots. We call this property the *port predicate* of the black box. The port predicate is expressed as a predicate on S, R, P, and the function N. The port predicate does not refer to the message sequences associated with the processes. The sequence of snapshots starting from the first snapshot that satisfies the port predicate is called a session. The processes cannot send or receive messages until the start of the session. It is up to the processes collectively or some external entity that has control over the processes to ensure that the port predicate is followed at all snapshots in a session.

If the port predicate is followed by the processes connected to the black box then the black box guarantees that it will follow certain rules in transferring messages in the send sequences to the receive sequences. This set of rules is called the *message predicate* of the black box. The message predicate is expressed as a predicate on the set P, and the message sequences M_s and M_r, and the function N. The message predicate defines how the messages received from the black box are related to the messages sent into the black box. Note that the processes in S on any particular snapshot may not be the only processes to have sent messages into the black box; there could be processes that were in the set S in a previous snapshot and that have sent messages to the black box. The same is true for the set R. The processes follow the port predicate and expect the black box to follow the message predicate. The black box expects the processes to follow the port predicate and guarantees the message predicate as long as the port predicate is followed.

Taken together, the port predicate and the message predicate completely define the operation of the black box. If we replace a black box by another that expects the same port predicate and follows the same message predicate the processes would not observe the change; the new black box also satisfies the communication requirements of the processes. We say that all the black boxes that follow the same port and message predicates

are of the same *communication type*. A communication type t is defined by the tuple $< P_t(\delta), M_t(\delta) >$ where δ is a snapshot and $P_t(\delta)$ is a predicate on S, R, and N that defines the port predicate and $M_t(\delta)$ is a predicate on P, M_s, M_r and N that defines the message predicate.

We explain how to use this black box model to define communication types by defining some well known communication types. The first communication type we define is the single source multicast with in-order delivery of messages with possible message drops. The source does not change over the lifetime of the multicast session and the source does not receive the messages. We denote this communication type by $ssmcast$ and define it as follows.

$$P_{ssmcast}(\delta) \triangleq |S| = 1 \wedge R \cap S = \emptyset.$$
$$M_{ssmcast}(\delta) \triangleq \exists_{q \in P} \, \forall_{p \in P} \, [M_r(p) \sqsubseteq M_s(q)].$$

We use the symbol $\sqsubseteq$ to denote a subsequence where $\alpha \sqsubseteq \beta$ indicates that α contains only the elements in β in the same order but may not contain all the elements of β.

The port predicate, $P_{ssmcast}(\delta)$, requires $|S| = 1$ on every snapshot. This guarantees the property that the source never changes because for the source to change we have to take the current source out of the send enabled set, S, and put the new source into S. However, this cannot be achieved without violating the port predicate since there is a snapshot that makes $S = \emptyset$. In the message predicate we do not have to specify that the send sequences on the receivers are empty. This is implied in the port predicate which insists that $S \cap R = \emptyset$. The message predicate must be understood in the context of the port predicate. This example shows how the black box model allows us to define the communication type precisely and concisely.

We do not use the set S or R in the message predicate. This is because these sets can be transient. For example, in $ssmcast$ the set R can change over time. A message sent in one snapshot may be delivered to a process that wasn't in R at that snapshot. M_s and M_r keep the whole history of the messages sent and received. Therefore, we use M_s, and M_r to define the message predicate.

The port predicate of $ssmcast$ does not mention N. If a port predicate does not use N, we consider that all the processes have the default value of $N(p)$, which is 1.

Note that $ssmcast$ is just a nickname for the above communication type. The actual name is given by the tuple $< P_{ssmcast}(\delta), M_{ssmcast}(\delta) >$. However, for convenience we use the nickname $ssmcast$ when we talk about that particular communication type. Any other arbitrary nickname can serve the same purpose. We use this convention throughout this thesis to refer to communication types.

We denote all-to-all multicast where all the processes can send and receive messages by $atoa$. $atoa$ is defined as follows.

$$P_{atoa}(\delta) \triangleq \mathbf{true}.$$
$$M_{atoa}(\delta) \triangleq \forall_{p \in P} \exists_{\alpha \in SendInterleave(P)} [M_r(p) \sqsubseteq \alpha].$$

Where,

$$SendInterleave(P) = \{\gamma : (\forall_{p \in P} [M_s(p) \sqsubseteq \gamma]) \wedge (\forall_{m \in \gamma} \exists_{p \in P} [m \in M_s(p)])\}.$$

This flavor of all-to-all multicast guarantees that the receivers get the messages in the same order as they were sent with possible gaps. However, it does not guarantee that all the receivers get the same message sequence. Different receivers may get messages sent by different senders in different order with respect to other senders. Note that the port predicate does not impose any restriction.

We define $gather$ as the communication type in which there is only a single recipient and all the other processes send messages to that recipient. If a message sent by a sender is received at the receiver all the messages up to and including that message have been received at the receiver in the same order as they were sent. The single recipient does not send any messages and the senders do not receive messages.

$$P_{gather}(\delta) \triangleq |R| = 1 \ \wedge \ S \cap R = \emptyset.$$
$$M_{gather}(\delta) \triangleq \exists_{p \in P} \forall_{q \in P} \forall_{m \in M_s(q)} [received(m, P) \Rightarrow \ to(m, M_s(q)) \sqsubseteq M_r(p) \ ].$$

The predicate $received$ is defined as follows:

$$received(m, P) \triangleq \exists_{p \in P}\, m \in M_r(p).$$

$to(x, \alpha)$ is a sequence consisting of elements in α up to and including the element x.

We define guaranteed message delivery as the guarantee that if a message is delivered then all the messages up to that message have been delivered. Since the black box is not capable of instant message delivery, guaranteed message delivery has to be expressed in the above manner. This is similar to the delivery guarantee provided by TCP, which in addition also provides ordering guarantee. TCP does not guarantee that all the bytes sent will be delivered; it guarantees that if a byte is delivered all the bytes up to that have been delivered.

$gather$ can be considered as the inverse of the $ssmcast$ where messages from a single source are received by all the other processes (However, note that there is no delivery guarantee in $ssmcast$). In the above definition of $gather$ the fact that the receiver does not send messages (and vice versa) is not explicitly stated in the message predicate since this is implied by the port predicate. Even if this fact is explicitly included in the message predicate it defines the same communication type since the message predicate must be interpreted in the context of the port predicate. However, omission of such details results in a more concise definition of the message predicate.

The communication type $anycast$ sends each message to any one of the receivers and there is always at least one process that is willing to receive messages. The senders do not receive messages and receivers do not send messages. $anycast$ is defined as follows.

$$P_{anycast}(\delta) \triangleq |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$

$$M_{anycast}(\delta) \triangleq Ordering(\delta) \wedge$$

$$\forall_{p \in P} \forall_{m \in M_s(p)} [\, received(m, P) \Rightarrow \forall_{w \in to(m, M_s(p))} \exists_{q \in P} (w \in M_r(q)) \,].$$

Where,

$$Ordering(\delta) \triangleq \forall_{p \in P} \forall_{q \in P} [SameOrder(M_s(p), M_r(q)) \,].$$

$SameOrder(\alpha, \beta)$ evaluates to true if the elements common to α and β occur in the same order in β as they do in α. $Ordering(\delta)$ ensures that all the messages received are received in the same order as they were sent.

This definition of $anycast$ ensures that if a message sent has been received at some receiver all the messages set by that sender up to that message have been received at some receiver. If a receiver gets more than one message from the sender they are received in the same order as they were sent.

The above definition can easily be modified to define a communication type similar to $anycast$, but that ensures that a message is received by exactly one receiver.

$$P_{anycast\_unique}(\delta) \triangleq |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$

$$M_{anycast\_unique}(\delta) \triangleq Ordering(\delta) \wedge$$
$$(\forall_{m \in M_s(p)} [\, received(m, P) \Rightarrow \forall_{w \in to(m, M_s(p))} \exists!_{q \in P} [\, (w \in M_r(q)) \,] \,]) \,].$$

The communication type $anycast\_amo$ that delivers a message to at most one receiver is defined as follows.

$$P_{anycast\_amo}(\delta) \triangleq |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$

$$M_{anycast\_amo}(\delta) \triangleq \forall_{p \in P} \forall_{m \in M_s(p)} [(\exists!_{q \in P} [m \in M_r(q)]) \ \vee (\forall_{r \in P} [m \notin M_r(r) \,]) \,].$$

There are no ordering or delivery guarantees in $anycast\_amo$.

Note that $anycast, anycast\_unique$, and $anycast\_amo$ have the same port predicate but different message predicates.

We define $manycast$ to be the communication type that sends a message to a set of any N receivers, where N is specified by the sender. If a message sent by a sender has been received at any receiver then $manycast$ guarantees that all the messages sent before that by the sender have been received at exactly N receivers. There is a set of senders distinct

from the set of receivers.

$$P_{manycast}(\delta) \triangleq (\forall_{s \in S}\, [N(s) \leq |R|]) \wedge\ S \cap R = \emptyset.$$

$$M_{manycast}(\delta) \triangleq\ Ordering(\delta)\ \wedge \forall_{p \in P}\, \forall_{m \in M_s(p)}\, [\, received(m, P) \Rightarrow$$
$$\forall_{w \in upto(m, M_s(p))} \exists!_{Q \subset P}\, [\, |Q| = N(p)\ \wedge\ \forall_{q \in Q}\, [w \in M_r(q)]\,].$$

Where, $upto(m, \alpha)$ is a sequence similar to $to(m, \alpha)$ but which excludes the last element in $to(m, \alpha)$.

$manycast$ is the first communication type that we defined to use the parameter N associated with each process. In fact, we introduced N to the black box model to handle $manycast$ and communication types similar to $manycast$. Note that if a process p does not specify $N(p)$ we consider that $N(p) = 1$. Later in this chapter when we discuss type conformance we show that the default value of $N(p)$ helps identifying conforming communication types, which otherwise would not have been considered conforming.

The communication type that receives messages from a certain number of senders, $somecast$, is defined as follows.

$$P_{somecast}(\delta) \triangleq\ \forall_{r \in R}\, [N(r) \leq |S|\ \wedge\ S \cap R = \emptyset].$$

$$M_{somecast}(\delta) \triangleq\ \forall_{p \in P}\, \forall_{\alpha \in subs(M_r(p), N(p))}\, \exists!_{Q \subseteq P}\, \forall_{m \in \alpha}\, \exists_{q \in Q}$$
$$[m \in M_s(q)\ \wedge\ |Q| = N(p)\ \wedge\ \forall_{q_1 \in Q}\, \exists_{m_1 \in M_s(q_1)}\, [m_1 \in \alpha]\,].$$

$subs(\beta, n)$, where β is a sequence and n is an integer, is a set of all contiguous subsequences of size n of β.

This definition of $somecast$ requires that each subsequence of $N(p)$ messages received at the process p come from $N(p)$ different senders. We did not define the behavior when a receiver has received less than $N(p)$ messages. It is simple to include the behavior for this case in the above definition, but we omitted this for the sake of the simplicity of the definition.

The communication types that we have defined so far do not use the set P in the definition of the port predicate. The communication type $gather\_mcast$, defined below, does.

$$P_{gather\_mcast}(\delta) \triangleq \exists_{p \in P} [p \in S \wedge p \in R \wedge \forall_{q \in P-\{p\}} [N(p) \neq N(q) \wedge \forall_{r \in P-\{p\}} [N(q) = N(r)]]].$$

$$M_{gather\_mcast}(\delta) \triangleq \exists_{p \in P} \forall_{q \in P-\{p\}} [N(p) \neq N(q) \wedge M_r(p) \sqsubseteq SendInterleave(P - \{p\}) \wedge M_r(q) \sqsubseteq M_s(p)].$$

The communication type $gather\_mcast$ uses $N(p)$ to identify a unique process. The port predicate states that there is a process with a unique number associated with it and all the other processes have a common number associated with them. This unique process is in both S and R. Other processes have the choice of being in S or R. The messages sent by all the other processes are received only at this uniquely numbered process in the same order as they were sent, but not all messages are received. The unique process does not receive its own messages. All the other processes receive a subsequence of messages sent by the unique process and they do not receive their own messages or messages from processes other than the unique process. Note that when there is only one process in P the message predicate is trivially true. When there are only two processes in P, and if they both are in S and R, we cannot identify a unique process, but still the message predicate can be followed. Once more than two processes are in P we can identify the unique process. The port predicate requires that all the processes other than the unique processes have the same number associated with them and this is apparent only when there are more than 2 processes in P.

The communication type $gather\_mcast$ uses $N(p)$ in a different way than $manycast$. In $gather\_mcast$, $N(p)$ is used to identify processes, while in $manycast$ $N(p)$ has the semantics of a number.

Finally we define one of the most simple communication types, $unicast$. We define

$unicast$ as follows.

$$P_{unicast}(\delta) \triangleq\ S = R = P\ \wedge\ |P| = 2.$$

$$M_{unicast}(\delta) \triangleq\ \forall_{p \in P}\, \forall_{q \in P}$$
$$[p \neq q \Rightarrow (\forall_{m \in M_s(p)}[\, received(m, M_s(p)) \Rightarrow to(m, M_s(p)) \sqsubseteq M_r(q)] \,)\,].$$

This definition of $unicast$ requires that if a message is delivered all messages up to that message are delivered and that they are delivered in-order. This is somewhat similar to what TCP guarantees.

Note that in the above definition of $unicast$ the port predicate requires that $|P| = 2$. This also means that the processes cannot change over time since once a process is in P it will be there for ever and the port predicate requires exactly two processes in P. Both processes must be in S and R in all the snapshots.

More on port predicates

The port predicate provides the connection between the implementors of the communication type and the "users" (users are discussed below). The port predicate provides information about how the communication type is used to the implementors and they can use this information to implement the message predicate. It is important that the port predicate of a communication type be meaningful to the message predicate. Take for example the following communication type nicknamed $bogus$.

$$P_{bogus}(\delta) \triangleq\ (\ \forall_{s \in S}\ N(s) > 1\ )\ \wedge\ S \cap R = \emptyset.$$

$$M_{bogus}(\delta) \triangleq\ Ordering(\delta)\ \wedge \forall_{p \in P}\, \forall_{m \in M_s(p)}\, [\, received(m, P) \Rightarrow$$
$$\forall_{w \in upto(m, M_s(p))} \exists!_{Q \subset P}\, [\, |Q| = N(p)\ \wedge\ \forall_{q \in Q}\, [w \in M_r(q)]\,].$$

The message predicate of $bogus$ is the same as the message predicate of $manycast$. However, the port predicate is different. The port predicate of $manycast$ ensured that there are as many receivers as the largest $N(p)$ of all the senders at all time. In $bogus$ there is

no such restriction on the size of the receiver set and because of this the message predicate cannot be followed in a meaningful manner. If there are no required number of receivers all the messages after the first one have to dropped

In this discussion we simply assumed that the processes follow the port predicate. In practice it is up to the programmers who write the code for these processes to make sure that the port predicate is followed at all times. However, the responsibility of ensuring the port predicate does not stop at the application programmers. This responsibility may also extend to the users who ultimately use the applications. For example, it is up to a "real" user to keep the server running at all times in a single source multicast session. Similarly, a user or set of users must keep at least N number of receivers running in $manycast$. Recall, that we do not assume a central control over the application components that join a session. We also do not require that the code for all the processes be written by a single programmer. The only commonality that we assume of the processes communicating over a session of a communication type is that they collectively follow a port predicate. The idea of the port predicate goes beyond the idea of an interface between programming elements (objects, layers, etc.) because it extends beyond the program text.

4.1.1 Communication type equivalence

We use first order predicate logic to define the port predicate and the message predicate. Infinitely many different formulae can be used to define the same port predicate and the same message predicate - in other words the same communication type. Similarly the same communication type can be given different nicknames since we do not propose to standardize these names. It is important that we are able to check whether two communication types are equivalent in all respects despite different formulae and nicknames. We define two communication types, t_1, t_2 to be equivalent if the following relation holds.

$$\forall_\delta \; P_{t_1}(\delta) \Leftrightarrow P_{t_2}(\delta) \wedge M_{t_1}(\delta) \Leftrightarrow M_{t_2}(\delta).$$

Take for example the communication type x.

$P_x(\delta) \triangleq\ |R| = 1\ \wedge\ S \cap R = \emptyset.$

$M_x(\delta) \triangleq\ \exists_{p \in P}\, \forall_{q \in P}\, \forall_{m \in M_s(q)}$

$$[\neg received(m, P) \vee received(m, P)\ \wedge\ (to(m, M_s(q)) \sqsubseteq M_r(p))\,].$$

It is trivial to show that $P_x(\delta) \Leftrightarrow P_{gather}(\delta)$ and $M_x(\delta) \Leftrightarrow M_{gather}(\delta)$. This proves that x and $gather$ are equivalent and x is simply another nickname for $gather$.

The fact that the same communication type can be defined by infinitely many different formulas does not hinder the communication type system. In fact, this allows the communication pattern needed by an application to be expressed in most natural way to the application. Once the required type is known it can be compared against the already known and available communication types to find a suitable implementation. However, we do not expect that process can be automated in general since predicate logic does not constitute a decidable system [11].

Note that type equivalence and type conformance (discussed below) are extra properties we observed in the black box model and are not central to our objectives in defining the black box model. However, these properties give credence to the black box model since it shows that the black box model is capable of capturing subtle properties of communication types.

4.1.2 Type conformance

Type conformance is a weaker relation than type equivalence. It allows us to identify communication types that can be used in place of another communication type even if they are not equivalent. We define type conformance as follows.

A communication type t_b conforms to communication type t_a if the port and message predicates of t_a and t_b have the following properties.

$$\forall_\delta\ P_{t_a}(\delta) \Rightarrow P_{t_b}(\delta) \tag{4.2}$$

$$\forall_\delta\ M_{t_b}(\delta)/P_{t_a}(\delta) \Rightarrow M_{t_a}(\delta) \tag{4.3}$$

Where $M_{t_b}(\delta)/P_{t_a}(\delta)$ stands for $M_{t_b}(\delta)$ in the context of $P_{t_a}(\delta)$.

This idea of type conformance was borrowed from Emerald [17, 78]. However, we do not claim that the communication types are related to type systems in programming languages and the black box model was developed with different objectives.

If t_b conforms to t_a then an implementation of t_a can be replaced with an implementation of t_b without any modification to an application that uses t_a. In fact, we show in Chapter 5 that our framework allows such replacements without any modification to the processes except to the process that takes the decision to do this replacement.

Take for example the communication types $ssmcast$ and $atoa$ that we defined before. $P_{ssmcast}(\delta) \Rightarrow P_{atoa}(\delta)$ is trivially true since $P_{atoa}(\delta)$ is defined as $\mathbf{true}$. In the context of $P_{ssmcast}(\delta)$, $SendInterleave(P)$ becomes the singleton set that contains the send sequence of the only sender. Therefore, in the context of $P_{ssmcast}(\delta)$, $M_{atoa}(\delta) \Rightarrow M_{ssmcast}(\delta)$. Therefore, $atoa$ conforms to $ssmcast$. An implementation of $ssmcast$ can be replaced with an implementation of $atoa$.

We can also show that $anycast$ conforms to $gather$. It can be shown that

$$P_{gather}(\delta) \Rightarrow P_{anycast}(\delta).$$

Since the only difference between the port predicates of $gather$ and $anycast$ is that in $gather$ there must be exactly one receiver and in $anycast$ there must be at least one receiver, it is easy to see that this is true.

In $anycast$ if a message is received at a receiver all the messages sent by that sender up to and including that message have been received at some receiver. Also, if a receiver gets more than one message from a sender they are received in the same order as they were sent. Under the port predicate of $gather$ there is only a single receiver and as shown above $P_{gather}(\delta) \Rightarrow P_{anycast}(\delta)$. Under the port predicate of $gather$ if a message is received all the messages from that sender up to that message must be received by the single receiver in the same order as they were sent and this is true for all the senders. This is the message predicate of $gather$.

$$M_{anycast}(\delta)/P_{gather}(\delta) \Rightarrow M_{gather}(\delta).$$

Therefore, $anycast$ conforms to $gather$. That is, we can transparently replace an implementation of $gather$ with an implementation of $anycast$ in an application that uses $gather$. $anycast\_unique$ also conforms to gather. $anycast\_unique$ guarantees that each message is delivered to exactly one receiver. Under the port predicate of $gather$ there is exactly one receiver and all messages are delivered to this receiver.

It seems counter-intuitive that $anycast$ conforms to $gather$ and not $gather$ conforms to $anycast$. However, the port predicate of $anycast$ allows more than one receiver and in that sense $gather$ is a more restrictive version of $anycast$. Therefore, $anycast$ conforming to $gather$ is the reasonable relation.

It can be shown that $manycast$ conforms to $anycast$. The port predicate of $anycast$ is

$$P_{anycast}(\delta) \triangleq \ |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$

Since N is not specified in the port predicate all the processes have the default value 1. Therefore, we can rewrite the port predicate as

$$P_{anycast}(\delta) \triangleq \ (\forall_{s \in S} \ [N(s) \leq |R|]) \wedge \ S \cap R = \emptyset.$$

This is the port predicate of $manycast$. Therefore,

$$P_{anycast}(\delta) \Rightarrow P_{manycast}(\delta).$$

$manycast$ guarantees that if a message, m, from a process p is delivered, each message up to that message is delivered to exactly $N(p)$ receivers. Under the port protocol of $anycast$, $manycast$ delivers each message up to m to exactly one receiver. Since the message m also has been delivered, $manycast$ delivers each messages up to and including m to exactly one receiver in the same order as they were sent. Since $anycast$ requires messages up to and including m to be delivered to at least one receiver,

$$M_{manycast}(\delta)/P_{anycast}(\delta) \Rightarrow M_{anycast}(\delta).$$

Therefore, $manycast$ conforms to $anycast$.

Note that the direction of the implication of Property 4.2 is the reverse of the direction of the implication in 4.3. This is similar to the contravariance property of the Emerald [17, 78] type system where conformance order of the arguments of the operations are the reverse of the conformance order of the types that have these operations. The direction of the implication of the port predicates ensures that the port predicate of the type to be replaced can be used with the new type. And Property 4.3 ensures that under the old port predicate the new message predicate provides the same behavior as the old message predicate.

The ability to switch an implementation of a communication type with an implementation of a conforming type is a useful feature. It extends the reusability of the implementation of the conforming type and also gives more options to applications. This is possible because of the division of the responsibility between the users and the implementation through the port predicate and the message predicate. As we mentioned before the responsibility of following the port predicate extends beyond the application programmers.

4.2 Limitations of the model

We identified three main limitations of the model that inhibit it from defining some interesting communication types.

1. We cannot define a communication type that guarantees the delivery of all messages sent.

2. Communication types allow the black box to drop all the messages.

3. Ordering between send and receive sequences of a process cannot be specified.

The black box cannot deliver messages instantly; it requires at least two snapshots to send a message and deliver it. Therefore, a message predicate that guarantees message delivery cannot be true in all possible snapshots. Therefore, we defined delivery guarantee

as: if a message is received all previous messages from the same sender up to that point have been received. While the inability to guarantee complete message delivery is a limitation, this is more closer to the real world situations.

The communication type $ssmcast$ illustrates the second limitation.

$$P_{ssmcast}(\delta) \triangleq |S| = 1 \wedge R \cap S = \emptyset.$$
$$M_{ssmcast}(\delta) \triangleq \exists_{q \in P} \forall_{p \in P} [M_r(p) \sqsubseteq M_s(q)].$$

The message predicate of $ssmcast$ guarantees that all the receivers get a subsequence of the messages sent by the only sender. Even if all the processes do not get any message this message predicate is satisfied; a black box that discards all the messages can satisfy the message predicate of $ssmcast$. Note that the message predicate of $ssmcast$ has the flexibility to deliver a message sent in one snapshot in a later snapshot without violating the message predicate.

Augmenting the black box with temporal logic can solve these two problems. The first problem can be solved by specifying the delivery guarantee as eventual delivery of messages. The current model can be extended by defining a function M_r^{∞}, which denotes the receive sequences at some undetermined time in future. However, this also does not force the black box to deliver messages since M_r^{∞} is always a future state. The current notion of guaranteed delivery is based on the observation of the messages sent and delivered so far and is not based on an expectation of a future state. In that sense, the current notion of guaranteed delivery is more appropriate to the real world situations. The second problem can be solved by specifying fairness; that is specifying that the state that delivers a message is reached many times. However, these are substantial changes to the black box model and further investigation is required to understand the full impact of such changes.

The model cannot describe a communication type that imposes an ordering between send and receive events of a given process. This is because the sending messages by a processes and the black box delivering messages to that process are two independent events. Therefore, we cannot talk about a message sent after receiving another message. An implication of this limitation is that communication types that provides certain causal ordering

cannot be defined. This can be solved by adding time stamps or sequence numbers to the messages sent and received. This imposes synchronization between sending and receiving messages and also adds further semantics to the messages; in the current model messages do not have any meaning other than that each message sent is globally unique.

Note that in this work we do not attempt to capture properties such as security, privacy, and quality of service of communication paradigms. We consider these as qualities of different implementations and we only capture what we consider as the functionalities of communication paradigms. While one can argue that certain properties that we set aside as qualities are the functionalities required by some applications we note that any abstraction has to limit itself to certain set of functionalities.

All these limitations are there because we attempted to keep the model simple. Our goal of introducing the model was to introduce the notion of communication types. There is room for improvement in the model. However, the current model with its limitations has served our purpose of presenting the concept of multiparty communication types.

4.3 Sessions of communication types

The communication type definitions are abstract and declarative. A communication type does not describe how a black box follows its message predicate or how the processes follow the port predicate; in other words it does not specify an implementation. To realize a black box as a concrete entity that real processes can use to send and receive messages it must be implemented. We describe one particular method of implementing communication types in Chapter 5 and we delay the discussion of implementing and using communication types until that chapter. We defined a session abstractly as a sequence of snapshots, but in practice a session is a runtime instance of an implementation of a communication type; an instance of an implementation of a black box of communication type t is a session of communication type t.

A session must be started by executing the implementation and we do not consider a session started until it is capable of following the port predicate. Take for example the

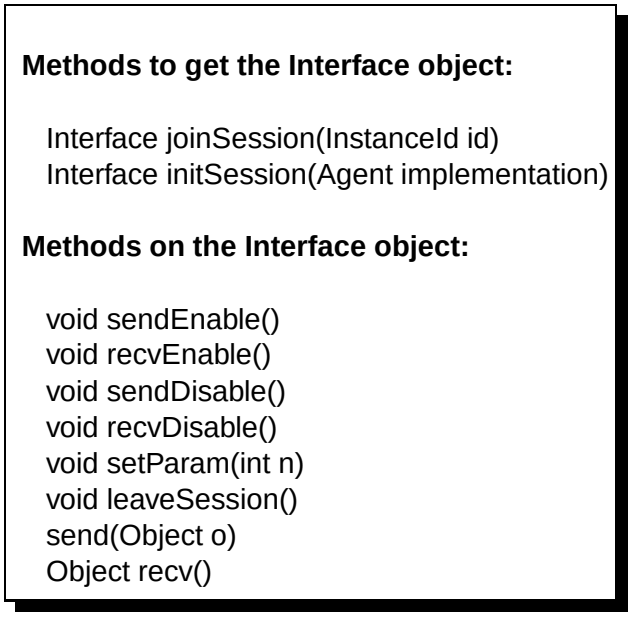

Figure 4.2: The application programmer's interface to communication types.

simple communication type of $unicast$. The port predicate of $unicast$ requires that there are exactly two processes connected to the black box and they both are send and receive enabled. It is impossible for both processes to join the session in the same snapshot. In practice, this is especially difficult considering the fact that there may not be a central controller over these processes. We do not consider a $unicast$ session as started until two processes join the session.

Once the session has started, then an implementation can assume that the processes follow the port predicate all the time. The message predicate must be guaranteed as long as the port predicate is followed. If the port predicate is violated we consider that the session ceased to exist and the behavior of the implementation is undefined from that point onwards.

4.4 Application programmer's interface

We now map this model to an application programmer's interface (API). The discussion so far has shown that the black box model is powerful enough to describe a large number of

communication types. Therefore we argue that it is reasonable to define an API based on the properties of the black box. We derive an API that can accommodate all communication types based on the actions that can be performed by processes on the black box.

There are several actions that can be performed on the black box that lead to a possible snapshot. These actions are external to the black box and taken by the processes connected to the black box. We map an API to each of these actions and that API is supported by all the implementations of communication types. In the following discussion we use a Java-like language binding to explain the API, but the API is not language specific. However, it may not be straight forward to map this API to different languages and different programming styles. As we mentioned in Section 3.2 these are problems associated with any API and we do not address these issues in this thesis. The complete API is given in Figure 4.2.

Any change of the set of processes P can lead to a snapshot. P can be changed by adding a process to P. We associate the following method with adding a process to the set P.

`Interface`[1] `joinSession(InstanceId id)`

`joinSession()` is called by a process that wants to be added to the set P and the call returns an `Interface` object. All the other methods are invoked on that `Interface` object.

`sendEnable()` adds the calling process to the set S. `recvEnable()` adds the calling process to the set R. Note that these two calls are not mutually exclusive. Similarly `sendDisable()` and `recvDisable()` remove a process from the set S or R respectively. `setParam(int n)` sets the value $N(p)$ associated with the process and if not called by a process that process is assigned the default value 1. Whether a process can call these methods depends on the port predicate. For example, in $ssmcast$ only one process is allowed to add itself to the set S. An implementation could make sure that once a processes added itself to the send set no other process is allowed into the send set. These five

[1]Not to be confused with the Java `interface` keyword

methods may be called before the start of any communication and if called after the start of the communication it is considered as the process leaving the sets S and R and joining as a new process. An implementation does not have to strictly follow this by actually forcing the process to leave and again join the instance. However, it has to be considered as those set of actions have taken place virtually.

`send(Object m)` called by the process p adds the message m to the sequence $M_s(p)$ as the last message in the sequence. For this call to succeed p must be in the set S. The message m is opaque to the black box. As far as the black box is concerned m does not have any semantic other than the fact that each message is unique across all the processes.

`Object recv()` called by a process p extracts the next message from the sequence $M_r(p)$ and returns it. That is, if a call to `recv()` returned $M_r(p)[i]$ then the next call to `recv()` returns $M_r(p)[i+1]$. The first call to `recv()` returns $M_r(p)[0]$. If there is no message in $M_r(p)$ that satisfies the above conditions then `recv()` blocks.

`leaveSession()` takes a process out of both S and R. Note that in the black box model a process cannot leave the set P. However, in practice by calling `leaveSession()` a process no longer participates in the session by sending or receiving messages. No other method on the `Interface` object can be called after calling this method.

All these methods require a black box of a given communication type. A black box is created by the first process of a session (the session leader) calling `initSession(Agent implementation)`. This call also acts as the `joinSession()` call for the session leader and the argument to the call is an implementation of a communication type.

The correct use of these methods for a session of a given communication type is defined by the port predicate of that communication type. For example, in $ssmcast$ only a single process can call `send()` method and other processes can call the `recv()` method.

The communication type system does not define any behavior on errors. Even though the model avoids handling errors, it is very important for an implementation to handle errors or inform the application. We delay the discussion on error handling until Chapter 5.

4.5 Related work

More often than not the communication paradigms we are interested in are presented with informal descriptions or with formal specifications described in natural language such as RFCs. However, there is a body of work that provides formal specification of reliable multicast protocols using diverse techniques. Livadas et al. [64] provide a formal specification for the Scalable Reliable Multicast (SRM) protocol using the timed I/O automata framework and also proves it properties. Lee et al. [59] model and verify a multicast protocol called MTP using Petri-nets. Creese et al. [29] use CSP [43] to specify and verify several protocols including a multicast protocol. Abbou et al. [1] formally specify and verify SPRM (Scalable Probabilistic Reliable Multicast) protocol using RT-LOTOS.

Renesse et al. [91] use ML to describe the set of communication protocols in Horus group communication system. Authors claim that the amenability of ML to analysis and verification using theorem provers like Nuprl [36] makes it a good language for reference implementations.

Lin [61] uses first order predicate logic to specify transport protocols. Even though Lin's work is not about defining multiparty communication it is interesting because of the use of predicate logic. A protocol is specified as a set of communicating state machines and the temporal relations between the state transitions are specified by time expressions.

None of these protocol specification techniques are suitable for our requirements. These protocol specification languages define operational semantics of protcols. In contrast communication types are declarative and define abstract ideas such as *single source in-order delivery multicast* and not a particular "protocol" to implement that idea. This is the reason to avoid the term "protocol" in our discussion. The term protocol has certain connotations and implies a set of actions. There are multiple protocols that can implement any given communication type. For example, there are several protocols to implement total order multicast. Use of a language such as CSP or LOTOS to describe a protocol leads to a description of the protocol in terms of set of operations. This means a commitment to an implementation. The declarative nature of communication types avoids such a commitment

to an implementation in terms of a particular protocol.

Livadas and Lynch [64] note the lack of precise definition of reliability in the descriptions of most reliable multicast protocols. They identify the need for a precise specification of a Reliable Multicast Service (RMS) that captures reliability semantics of several reliable multicast protocols. RMS defines the reliability independent of the protocols that implement it. However, RMS itself is defined using the timed I/O automata and this leads to a definition that gives a set of actions, which can be considered as a meta protocol; a commitment to a particular set of protocols. Furthermore, this work only tries to capture semantics of some of the reliable multicast protocols and not communication paradigms in general.

The Script [37] abstraction is a tool to define a pattern of communication between fixed set of processes. A Script consists of set of *roles*, set of *data parameters*, and a *body*. The *body* gives a sequence of basic communication among the *roles* and expressed using a concurrent programming language like CSP or Ada. This commits the Script to a particular implementation. The purpose of the Script is to hide the implementation details in the body, but there is no mechanism in the Script itself to define the pattern without the body; the name of the Script by itself does not describe the communication pattern.

Roman et al. [83] present a framework for spatiotemporal communication. This is a parallel work to ours. They also recognize the appearance of new communication paradigms and the need to precisely define them. This framework defines communication paradigms in terms of the identity and location of the recipients and the time; that is, a message is delivered to a particular node in a particular location at a particular time. A set of predicates on the identity, location, and time defines the communication paradigm. Such a definitions are suitable for communication paradigms proposed for sensor networks and mobile networks. Authors give example definitions of multicast, Mobicast [45], Geocast [71] and Speed [41]. The inclusion of time and location parameters enables Roman et al.'s framework to capture some communication primitives that are not captured in our communication type systems. However, their framework has a limitation because the specification is given with respect to

a single message and hence cannot capture inter-message properties such as message ordering. Also they do not provide concepts similar to type equivalence and type conformance and do not attempt to map an API to the communication paradigms they define. This work is the closest in spirit to our work on communication types.

Liu et al. [63] presented the idea of *protocol switching*, which is somewhat similar to our notion of communication type equivalence. The switchable protocols are identified with respect to several predefined *meta properties* that are preserved under a particular runtime switching mechanism for the Ensemble and Horus systems [14]. A *switching protocol* that encloses switchable protocols is built and the upper layer uses this switching protocol. The switching protocol decides on which of these switchable protocols to be used and hides this decision from the upper layers. Our notions of communication type equivalence and conformance are not defined with respect to any switching mechanism and these are inherent properties of communication types independent of any particular implementation.

4.6 Contributions

We presented the concept of multiparty communication types. We showed that it is possible to name communication paradigms precisely using communication types. This solves the first issue we discussed in Chapter 3. We also presented the concepts of type equivalence and type conformance for multiparty communication types. The idea of type conformance allows applications to replace an implementation of a communication type with an implementation of a different (but conforming) communication type.

We also showed that it is possible to develop a common API that can cater to all the communication types. This solves the second issue we discussed in Chapter 3. A communication type completely defines the responsibility of the implementors of that type and the users of any implementation of that type. The idea of the port predicate transcends the program text and extends into the users of applications that use the communication type.

We used an abstract model to introduce the concepts of communication types, type equivalence, and type conformance. As we discussed before this model has several limita-

tions. We used this model as a vehicle to describe the concept of multiparty communication types and we do not claim that this is the best model for that purpose. Even with its limitations the model was able to describe large number of communication types. Developing a comprehensive model to describe communication types requires extensive research in its own right. However, we believe that this somewhat constrained model has served its purpose in highlighting the usefulness of the concept of communication types.

In the next chapter we will present the complete design of the MayaJala framework that supports the development and use of communication types based on the insight gained in this discussion combined with practical considerations.

Chapter 5

The Architecture of MayaJala

The communication type system provides solutions to two of the issues that we identified in Chapter 3; it provides a method to uniquely identify communication types and provides a generic API. In this chapter we address the third issue by designing a middleware system, MayaJala, that supports the implementation and deployment of communication types.

The communication type definitions are declarative. A communication type does not indicate how it can be implemented and only defines what is to be expected of an implementation. This opens up a vast design space in implementing communication types. Implementing any one of these communication types over the wide area Internet is a complex and specialized task. However, using them in applications should not be a complex task.

We show how the definitions of communication types can be used to clearly separate the tasks of implementing communication types from using them in applications. Unlike the communication type system, this separation of concerns is not just an abstract idea and our prototype demonstrates how it is used in practice. All the application components must agree on one particular implementation of a communication type to be used for a given session. MayaJala also shows that different implementations can be used for different sessions without any coordination or central control over the independently developed and distributed application components. This mechanism wasn't evident from the definitions of

communication types, but required to make the implementations widely usable.

We start this discussion by showing how the tasks of implementing communication types and using them in applications are assigned to two different group of programmers. Then we present the design of MayaJala from the point of view of those two groups of programmers.

5.1 Division of responsibilities

A communication type definition has two parts; the port predicate and the message predicate. It is the task of the black box to follow the message predicate and deliver the messages. It is the responsibility of the processes to follow the port predicate. In practice the processes are the applications (different applications or components of the same application) that communicate over a session of a communication type. They must be implemented by *application programers* in such a way that they follow the port predicate. Similarly we need an implementation of the black box that follows the message predicate under the port predicate; this is done by *type programmers*.

In Chapter 4 we mapped an API to the abstract model. The API describes the actions that the processes can take. Application programmers use this API to develop the applications that follows the port predicate. Note that the correct use of the API is defined by the port predicate of the communication type. The application programmers do not have to deal with the message predicate. Of course when selecting a communication type for an application the designers must be aware of the complete communication type—both port and message predicates. Once that decision is taken, application programmers can concentrate on making sure that the port predicate is followed by using the API correctly. Note that the port predicate extends beyond the program text as we explained in Chapter 4 and application programmers by themselves may not be able to completely enforce the port predicate. The small number of methods in the API is an indication that using a communication type in an application is not an overly complex task.

The task of type programmers is much more complex than the application program-

mers. As in the case with API there is no simpler set of methods that type programmers can use. Type programmer cannot concentrate just on the message predicate since the message predicate is interpreted under the port predicate. Type programmers can assume that the port predicate is followed and can make use of that information to simplify the implementation. For example, in $ssmcast$ the knowledge that there is only a single sender can be used to considerably simplify the implementation. Therefore we say that type programmers implement a communication type and not just the message predicate. There is a vast design space to be explored in implementing any given communication type. It is the task of the type programmers to explore this design space to implement communication types and to optimize them for different metrics such as bandwidth, latency, and robustness. This requires certain skills and knowledge that may not be expected of the users of these implementations. MayaJala provides facilities to simplify this task.

The design of MayaJala demonstrates how this clear division of tasks is achieved in practice. MayaJala provides support to use several communication types in an application at the same time independent of their implementation details, thus separating the task of application programmers from the type programmers. MayaJala also allows components of distributed applications to be developed separately, possibly by different programmers. This makes MayaJala more amenable to distributed applications on the Internet which have independently developed components, such as the different instant messaging clients, communicating with each other. In Chapter 2 we note that communication paradigms such as multicast have been implemented using application level overlay networks over the Internet. Note that as we mentioned in Chapter 1, the objective of this work is to support the implementation, deployment, and use of communication types over the Internet. MayaJala provides facilities to build overlay networks so that type programmers can use these facilities to implement and deploy communication types.

First, we present MayaJala from the perspective of application programmers and type programmers and then we present the overall operation of the system in detail. Application programmers are shielded from the complexity of the system and the implementation

```
public class TestServer {
  public static void main(String[] args) {
      MayaJala mjinstance = new MayaJala();
      MJInterface iface = mjinstance.initSession(new MJTestAgent());
      iface.sendEnable();
      System.out.println(iface.getNetId());
      while(true)
          iface.send(new String("Hello World"));
  }
}
```

Figure 5.1: An example server.

details of communication types. Therefore, application programmers have the most simple view of the system. This simplistic view also makes it a good starting point for describing the architecture of MayaJala.

We built a prototype of MayaJala using the Java programming language. The architecture of MayaJala is described with respect to this reference implementation. While the prototype heavily uses Java specific features, the design of MayaJala does not depend on language specific features. However, it may not be straight forward to port this prototype implementation to other languages.

5.2 Application programer's perspective of MayaJala

We use two example applications that use MayaJala and communication types to illustrate the application programmer's perspective of communication types and MayaJala. The first example is a very simple application that uses $ssmcast$ for communication. Even through it is simple, we also use it to show how the powerful concept of type conformance is used in practice.

The code listing in Figure 5.1 shows a multicasting server that repeatedly multicasts a message using the communication type $ssmcast$. There is a corresponding client application that repeatedly receives the messages (Figure 5.2). All the applications that

```
public class TestClient {
  public static void main(String[] args) {
      MayaJala mjinstance = new MayaJala();
      MJInterface iface = mjinstance.joinSession(new MJNetId(args[0]));
      iface.recvEnable();
      while(true)
          System.out.println("MayaJala: recv" + iface.recv());
  }
}
```

Figure 5.2: An example client.

use communication types implemented on MayaJala require an instance of MayaJala. Irrespective of the number of communication types used in the application, an application only needs a single instance of MayaJala. The server first creates a MayaJala instance and calls the `initSession()` method of the MayaJala instance to start an $ssmcast$ session. `MJTestAgent` is an implementation of $ssmcast$ and an object of `MJTestAgent` is passed as the argument to the `initSession()` method. We introduced this method in Chapter 4; it is the API call to create a new black box or a session of a communication type. The `initSession()` method returns an *interface* object. This interface object provides the API discussed in Chapter 4. The session gets a unique identifier and the `getNetId()` method of the interface object returns that identifier. The server simply prints the textual representation of the session identifier to the terminal in this example. Then the server calls the `sendEnable()` method of the interface object to send enable the interface and uses the `send()` method to repeatedly send a message on the $ssmcast$ session. Note that according to the port predicate of $ssmcast$ there is only one sender and there must always be exactly one sender. Therefore, the server must be started before any clients.

All the methods defined on the interface object are the API methods we described in Chapter 4. The interface object provides the interface to the session. It is up to the application to follow the port predicate and use the interface object accordingly.

The client code is given in Figure 5.2. The client also creates a MayaJala instance.

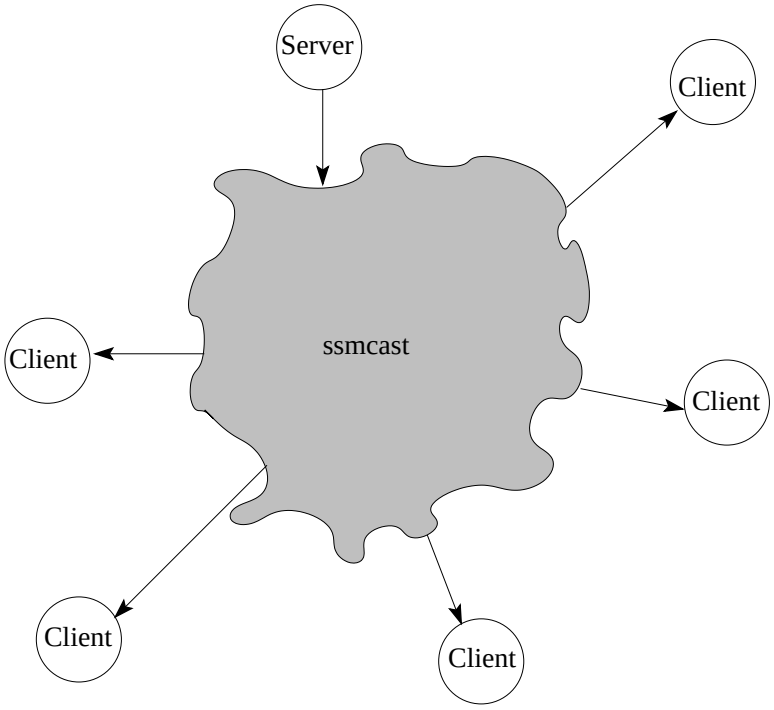

Figure 5.3: The application programmer's view of $ssmcast$.

The client gets the textual form of the unique identifier of the $ssmcast$ session on the command line. It uses that to join the session and, when successful, gets an interface object. Note that the client does not have any idea of the implementation of the communication type. In fact, the client does not even have to know which communication type is being used. The client only has to know the identifier of the instance and the port predicate of the communication type. According to the port predicate of $ssmcast$, all nodes, other than the source, can only call `recv()` and the client calls the `recvEnable()` method. The mechanism of using the correct implementation of the communication type is completely hidden from the client application and we discuss this mechanism later when we discuss the architecture of MayaJala in detail.

There may be many such clients that have joined the $ssmcast$ session. The port predicate of $ssmcast$ allows an unlimited number of processes that only call `recv()` on the interface, but it only allows one process that uses `send()` (a server). The application

programmers view of this scenario is shown in Figure 5.3. The "cloud" represents the $ssmcast$ session.

We do not assume that the client code is same at all the clients. The client code can be written by different programmers and may have different functionalities at different nodes. In the above example, a client on one node may simply display the message and on another node it may process the message and multicast it to another set of nodes. The only commonality that we assume of these clients and the server is that they all communicate over a given communication type and in addition to following the port predicate they all agree on an application specific message predicate. We do not require the clients only to communicate with the same server. It is simple for clients to join another session and another server without modifying the code. These servers also may have different functionalities and could be written by different programmers.

It is simple to change the implementation of the $ssmcast$ used in this application. Only the server has any knowledge of the implementation of the communication type. To use a different implementation of $ssmcast$ only one line in the server code has to be changed. For example, we could use `MJRandomTreeMcastAgent` (another implementation of $ssmcast$) in place of `MJTestAgent` by simply changing the corresponding line in the server code. The clients do not have to be aware of this change. The clients can join different sessions that use different implementations. It is possible to allow for the change of implementations of communication types without changing the application code at all by adding a level of indirection. However, we have not implemented this feature in this prototype of MayaJala.

This application can also use an implementation of a different communication type, $atoa$ (`MJAlltoAllMcastAgent` is an implementation of $atoa$). This is possible because $atoa$ conforms to $ssmcast$. This means under the port predicate of $ssmcast$, the behavior of $atoa$ is equal to that of $ssmcast$. In the above example there is only one sender and all the others are receivers. If we replace the implementation of $ssmcast$ with an implementation of $atoa$, the application still works correctly since $atoa$ allows any process to

be a sender or a receiver or both. The only change required to use a conforming communication type is same as the change required to use a different implementation of the same type.

The ability to use different implementations of a given communication type increases the *option value* [5] of the application. The property of communication type conformance increases the option value by allowing applications to use implementations of communication types that conform to the communication types already in use.

The next example uses two communication types, $anycast$[1], and $unicast$. In this example there are servers that receive messages on an $anycast$ session and there are clients that send messages on the same session. The first server starts an $anycast$ session and `recvEnable()` the interface (Figure 5.4). It prints the textual form of the session identifier. Subsequent servers gets this session identifier as a command line argument and they simply join the session and call `recvEnable()` on the interface object for this session. This enables them to receive messages on the $anycast$ session. All servers go into a receive loop. Each message a server receives over the $anycast$ session is a session identifier of a $unicast$ session. The server creates a worker thread to carry on a conversation on this $unicast$ session (Figure 5.5). The worker thread joins the $unicast$ session by calling the `joinSession()` method of the same MayaJala instance that it shares with main server thread. There is only one MayaJala instance per JVM instance in our implementation and all the threads use this instance to join and create sessions. The `joinSession()` method may take an arbitrarily long time to return and that is one reason to create a thread to do this; another reason is that while the conversation takes place over the $unicast$ session, the server can continue on receiving messages.

The client (Figure 5.6) also joins the $anycast$ session. However, instead of receive-enabling the interface each client send-enables the interface. The client creates a $unicast$ session and sends the session identifier of this newly created session over the $anycast$ session. The client then creates a thread (Figure 5.7) to carry out a conversation on this

[1]The actual implementation of this example uses $anycast\_amo$ and not $anycast$. However, we describe this example in terms of $anycast$ for the sake of simplicity.

```
public class AnyServer {

  public static void main(String[] args) {
    MayaJala mjinstance = new MayaJala();
    MJInterface any;

    if(args.length == 0)
    {
      any = mjinstance.initSession(new MJTrivialAnycastAgent());
      any.recvEnable();
      System.out.println(any.getNetId());
    }
    else
    {
      any = mjinstance.joinSession(new MJNetId(args[0]));
      any.recvEnable();
    }

    while(true)
    {
      MJNetId uniId = (MJNetId)any.recv();
      new Thread(new Worker(mjinstance, uniId)).start();
    }
  }
}
```

Figure 5.4: The use of $anycast$ and $unicast$ together (anycast server).

$unicast$ session.

The application programmer's view of the $anycast$ and $unicast$ sessions in this example is shown in Figure 5.8. Even though this example has a complex communication structure, the application programmer's view of the communication is simple; the application only sees different communication sessions and how the messages are delivered is irrelevant to the application programmers.

The above two example applications do not serve any useful purpose. However, they are representative of real world applications in the sense that they capture the essence

```
public class Worker implements Runnable {
 MayaJala mjinstance;
 MJNetId uniId;

 public Worker(MayaJala mj, MJNetId uniId) {
   this.mjinstance = mj;
   this.uniId = uniId;
 }

 public void run() {
   MJInterface uni = mjinstance.joinSession(uniId);
   uni.send(new String("From the worker"));
   uni.leaveSession();
 }
}
```

Figure 5.5: The worker thread.

```
public class AnyClient {

 public static void main(String[] args) {
   MayaJala mjinstance = new MayaJala();
   MJInterface any = mjinstance.joinSession(new MJNetId(args[0]));
   any.sendEnable();

   while(true)
   {
     MJInterface uni = mjinstance.initSession(new MJUnicastAgent());
     any.send(uni.getNetId());
     new Thread(new ClientHandler(uni)).start();
   }
 }
 }
```

Figure 5.6: The use of $anycast$ and $unicast$ together (anycast client).

of communication in some real world applications. For example, the simple multicasting server and the clients are representative of real world applications like multimedia streaming

```
public class ClientHandler implements Runnable {

  private MJInterface uni;
  public ClientHandler(MJInterface uni)
  {
    this.uni = uni;
  }

  public void run()
  {
    System.out.println(uni.recv());
    . . .
    uni.leaveSession();
  }
}
```

Figure 5.7: The client thread.

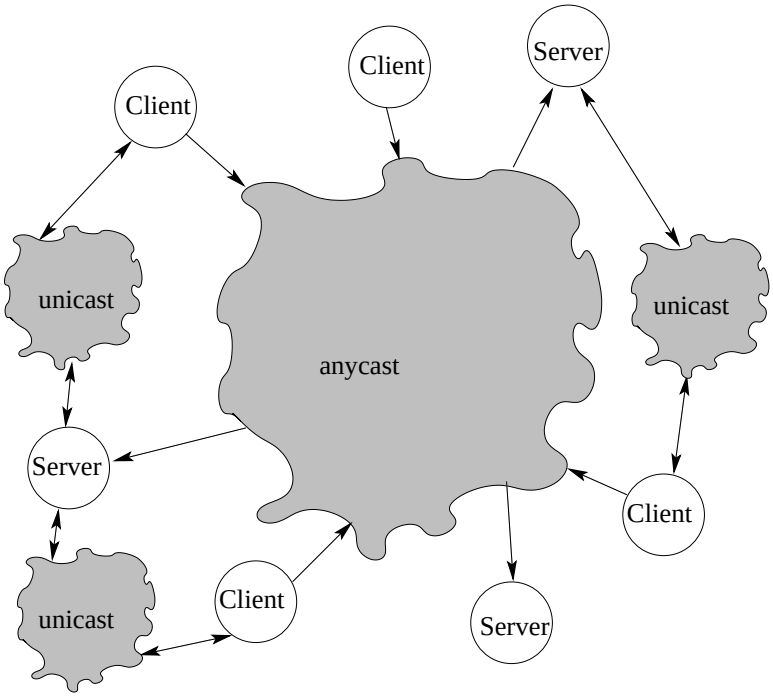

Figure 5.8: The application-programmer's view of $anycast$ and $unicast$ sessions.

applications, and stock market tickers.

5.2.1 The abstract model and the Java language bindings in MayaJala

Even though we used communication types such as $ssmcast$, $anycast$, and $unicast$ to describe the communication in the above example programs, these communication types do not appear in the Java code. This is because there is no programming language representation of communication types. Although we claim that `MJTestAgent` implements the communication type $ssmcast$ this fact is not embedded in its implementation. The identifiers given to implementations are arbitrary and should not be taken as indicative of the communication type[2].

Even if we were able to encode the communication types in the implementations and were able to describe the required communication type in the Java code, we do not expect that it is possible to do any form of "type checking". The reason for this is the use of predicate logic to define communication types; predicate logic does not constitute a decidable system [11]. That is, there is no mechanical procedure that can decide whether two communication types are equal for all possible combinations of communication types. However, in the most trivial case where two implementations of the same communication type have the same formula representing the communication type embedded in them, verifying this reduces to simple string matching. Implication of this is not as trivial as it seems. In practice we expect there would be large number of implementations of existing communication types rather than appearance of large number of new communication types frequently. New implementations of a communication type appear because they have some advantage over the existing implementations. It is reasonable to assume that the implementors would embed the same formula as the existing implementation in the new implementations if they want to claim that it is better than the existing implementation; to claim superiority they have to refer to the existing implementations and obvious choice is to use the same formula to describe the communication type. At this stage this is simply a reasonable assumption

[2]Note that the nicknames such as $ssmcast$ given to communication types are also meaningless without their formal definition.

and only the wide spread use of communication types can provide evidence to verify this.

While it is desirable to have a programming language representation of communication types, even without it the communication type system provides a powerful tool to the application designers. This is similar to design patterns [38]. Design patterns also do not have programming language representations. The collection of design patterns is a tool available to application designers to simplify the application design process. Similarly the communication type system simplifies the design of distributed applications with complex communication patterns. The communication type system is more powerful than design patterns because communication types are mathematically defined so that tools of mathematics can be used to manipulate them. The concept of type conformance is a byproduct of this mathematical definition of communication types.

While we expect that the processes to follow the port predicate collectively, it is possible to easily enforce the port predicate in some cases. For example, in $ssmcast$ the sender cannot receive messages and the receiver cannot send messages. This can be checked locally and the interface returned by `MJRandomTreeMcastAgent` enforces this by throwing an exception if the port predicate is violated in that manner. And in this case the agent also make sure that the interface returned to the session leader is only send enabled and on other nodes it is only receive enabled. This also ensures that there is only a single sender and that sender is in the session from the very beginning.

Note that the `AnyServer` (Figure5.4) and `AnyClient` (Figure 5.6) use separate threads to handle each $unicast$ session. Separate threads are used because of the blocking semantics of the API calls. In this prototype version of MayaJala we did not extend the API derived from the abstract model. Therefore, none of the API calls allow timeouts or nonblocking semantics. In practice such an extended API may be of use. A `recv()` call on the interface object blocks until there is a message to be received. While the `send()` calls do not block usually there is no guarantee that they do not block in general. The simple interface that we derived in Chapter 4 does not mention any blocking or non blocking semantics for the `send()` while the API derived from the black box model requires `recv()`

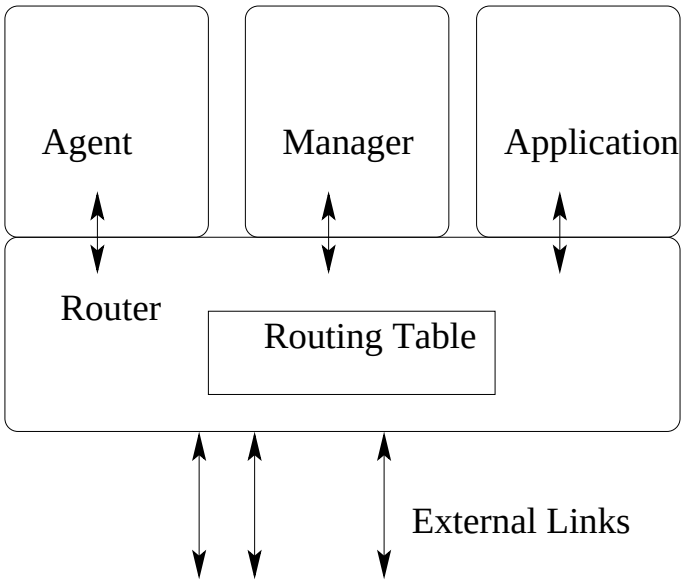

Figure 5.9: The type programmer's view of MayaJala.

to return with a message; this implies blocking until a message is available. We keep this simple interface and do not extend it to include `send()` and `recv()` calls with timeouts. However, such an extension is certainly possible.

5.3 The type programmer's perspective

Recall that the example server (Figure 5.1) creates an object of type `MJTestAgent` and passes it to the `initSession()` method of the MayaJala instance to start a $ssmcast$ session. `MJTestAgent` is an example of an implementation of a communication type—`MJTestAgent` implements $ssmcast$. The implementations of communication types are called *agents*. All the agents are subclasses of the abstract class `MJAgent`, which implements the Java `Runnable` interface. Type programmers implement agents.

The MayaJala instance that the server in Figure 5.1 creates has the architecture depicted in Figure 5.9. This is the agent's view of the MayaJala architecture. The application in Figure 5.9 is any arbitrary application that uses MayaJala for communication; an application similar to the example server in Figure 5.1. This is the application that created the

MayaJala instance. The agent is an implementation of a communication type as described above. We describe the function of the manager shortly. All these components run in separate threads and each has a duplex link to the router. At this point in the discussion it is sufficient to say that all the agents, the application, and the manager communicate with each other using messages with the router providing the necessary plumbing.

This discussion is presented through the perspective of type programmers and hence from the perspective of agents. The example server and example client in Figure 5.1 and Figure 5.2 are also used, together with code segments from an example agent, to explain how agents are built and deployed.

5.3.1 Agent's life cycle

On the session leader (example server) the agent's life cycle has three phases.

1. Creation: An agent is created when an object of the type `MJAgent` is created. This is the object passed into the `initSession()` method in the session leader. This agent object is passed on to the manager of the MayaJala instance.

2. Initialization: The `initSession()` method passes the `MJAgent` to the manager and the manager initializes it by supplying it with links to the router and a unique session identifier. The number of links required is discovered by querying the agent.

3. Activation: Note that all agents are runnable objects. The manager creates a thread and starts the agent on it. The agent goes into its own initialization phase. The first action an agent takes at this point is to register a clone of itself with the manager. This clone is stored in a table, keyed by the session identifier, in the manager. The purpose of this will be clear when the process of joining a session is described. The clone registration process is common to all the agents and this action is implemented in `MJAgent`. After the clone registration, the superclass calls `initProcess()`, which subclasses could extend to implement their own initialization functionality. After that the agent goes into an event handling loop. The agent

gets messages from the link to the router and it also sends messages to other components in the same MayaJala instance and also to other MayaJala instances through the router. After activation the agent must send an `MJInterface` object to the application. This `MJInterface` object is the one that is returned by the call to `MJInitSession()`.

On a non session-leader, for example the client in Figure 5.2, there is no creation phase. The example client also creates a MayaJala session, which has the same structure as the MayaJala instance on the session leader. Instead of calling `initSession()` the client calls the `joinSession()` method and passes it a session identifier (a `MJNetId` object). There is no indication of the type of the agent to be used for this session. The session identifier is passed to the manager and instead of a creation phase the process enters into a resolution phase. The session identifier encodes the manager on the session leader as the default resolver for this session. The manager on the client contacts the manager on the session leader to resolve the session identifier to an agent. Recall that on the session leader the agent registers a clone of itself with the manager. The manager on the session leader returns a copy of this agent object to the manager on the client. After downloading the agent, the manager initializes the agent as above, but in this case the agent only gets a link to the router; the agent already knows about the session identifier and the manager on the client does not have to generate a new session identifier. The activation phase is same as on the session leader.

Note that it is also possible for the managers on MayaJala instances other than the session leader to act as the resolvers for a session. The managers on the nodes that are participating in the session are the obvious choices, but any other manager also can act as a resolver. The prototype implementation has the facilities to allow this, but it still requires the manager on the session leader to act as the default resolver. This approach is taken for the sake of simplicity of the prototype implementation. However, it is possible to generalize this. This issue is not specific to MayaJala. Note that DHTs such as Pastry [84] and application level overlay networks such as Narada [27] also require a node to know an

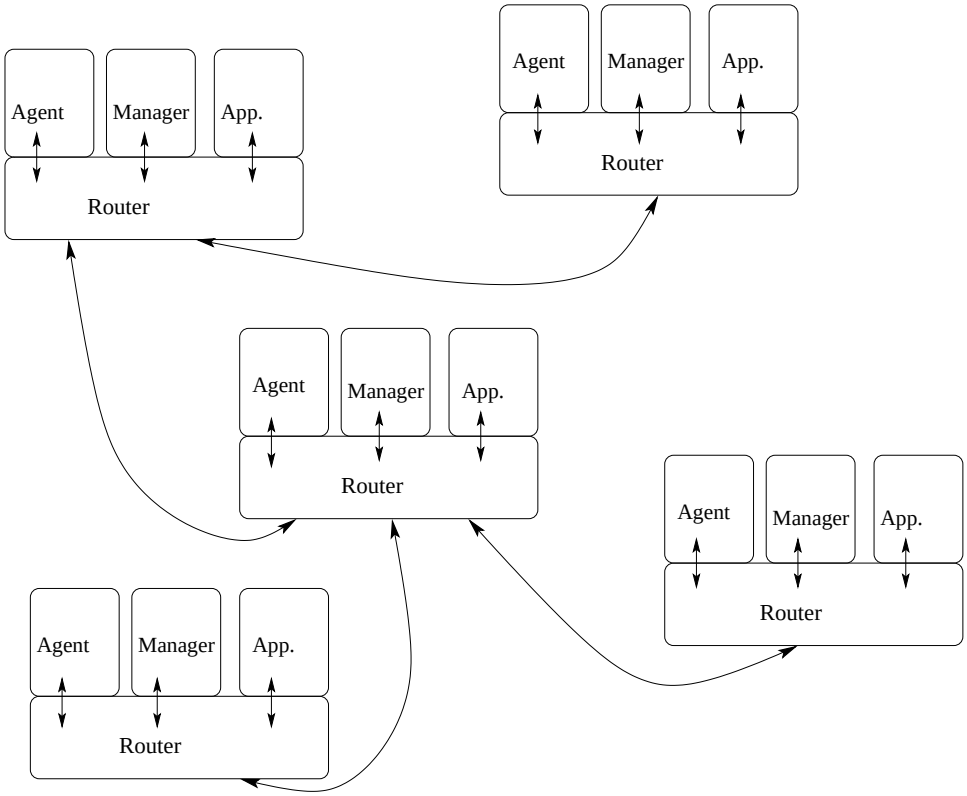

Figure 5.10: The type programmer's view of the overlay network.

existing member on the network to join the network.

The cloned agent that is started on the client usually has information on how to contact the session leader or any other agent currently in the session. These agents form a network; an overlay network on top of the IP network. The type programmer's view of this overlay network is shown in Figure 5.10. Usually the agents do not form a fully connected topology. In such a situation agents cannot communicate directly with each other and have to route messages over the overlay network. It is the task of each agent to set up the routing table in the router on its own MayaJala instance. The agent also inserts rules into the routing table to guide messages between the components in its own MayaJala instance. Some example routing rules are discussed shortly.

5.3.2 The code structure

```
public class ExampleAgent extends MJAgent {
  ...
  public MJAgent getClone();

  public void initProcessing() {...}

  public void processData(MJData msg){...}

  public void processExternalLink(MJReplyExternalLink msg){...}

  public void processLinkDown(MJReplyLinkDown msg){...}

  public void processReminder(MJReplyReminder msg){...}

  public void processLinkReport(MJReplyLinkReport msg){...}
  ...
}
```

Figure 5.11: An agent.

Figure 5.11 shows the code structure of a typical agent. All the agents extend `MJAgent`, which implements an event loop that reads messages coming from the router and calls appropriate methods for each message type. It is up to the subclasses to implement these methods if they are interested in these messages. There are two main classes of messages that an agent gets; `MJReply*` and `MJData`. The agents use `MJCommand*` messages to get services from the router. For example, to request that router create an external link to another MayaJala instance, the agent sends `MJCommandCreateExternalLink` message. The `MJReply*` messages are the router's replies to these commands. For each such `MJReply*` message there is an associated method that an agent must implement to handle the message, if it is interested in that type of messages. In Figure 5.11 `processExternalLink()` is the method that should be implemented to handle the reply to the command that asked the router to create an external link.

To handle incoming `MJData` messages the agent implements the `processData()` method. `MJData` is a generic message type provided for the sole use of the agent's overlay network. There are two subclasses of `MJData`; namely `MJDataBulk` and `MJDataLite`. The serialization of `MJDataBulk` is optimized for bulk data (this is described in Section 5.4.1) and `MJDataLite` is used for small messages. Both classes have an identical structure and we simply use `MJData` for the purpose of this discussion. The structure of an `MJData` message is shown in Figure 5.12. Agents use `MJData` messages as a means of sending application data as well as the control messages for the overlay network. `MJData` is structured to provide this flexibility. The router uses only the header fields in an `MJData` message to match against the routing rules. The `MJNetId` field carries the session identifier. The router changes the `in_link` field to indicate the incoming link and decrements the `TTL` on each hop. However, it does not attach any other semantics to the header and it completely ignores the payload. The agent has the complete freedom to use the type, source, and destination fields as well as the payload for its own purposes. Agents can decide on how the addresses (source, destination) are allocated in the network as well as the type of messages and their meanings to the network. Once it has decided on the addressing scheme and the types of messages in the network, an agent can decide how the messages are routed in the overlay network. As far as it is concerned, an agent may assume that it is the only agent that operates on the router; the router keeps each overlay network completely isolated from the others. Agents install routing rules into the router to guide `MJData` messages on their overlay networks. An agent could inject routing rules that are based on the incoming link, the source, the destination, and the message type into its own routing table in the router.

The `initProcessing()` method is the place to put the code that must be executed before an agent goes into the event processing loop. When an agent is activated the `run()` method of the `MJAgent` gets the control and it calls `initProcessing()` before entering the event loop. We discuss the `initProcessing()` method in Section 5.3.3 using an example agent. Agents must also implement the abstract method `getClone()`.

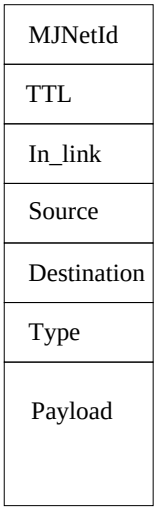

Figure 5.12: An `MJData` message.

The superclass, `MJAgent`, calls `getClone()` to get the clone of an agent and sends it to the manager to register it. This is done in the activation phase as described in Section 5.3.1.

5.3.3 An example agent

We use an agent, `MJRandomTreeMcastAgent`, that implements $ssmcast$ to describe the methods that an agent must implement. `MJRandomTreeMcastAgent` builds a multicast delivery tree rooted at a single source. A new node sends a `JOIN` message to the session leader to join a session. The session leader, if it has fewer than the maximum number of children that it can take, invites this new node by sending an `INVITATION` message. It also sends an `INVITE` message down the tree requesting other nodes to send invitations to this new node. The nodes that get the `INVITE` message also send an `INVITATION` message to the new node if they do not have the maximum number of children. The new node selects the first `INVITE` message it gets and becomes a child of the node that sent the invitation by sending an `ACCEPT` message. If a node is disconnected from its parent it sends a `JOIN` message again and follows the same procedure.

This agent is presented under several headings; building the clone, initialization before the event loop, handling replies from the router, processing data on the overlay, and

```
public MJAgent getClone() {
  return new MJRandomTreeMcastAgent(myNetId());
}
```

Figure 5.13: The `getClone()` method of the `MJRandomTreeMcastAgent`.

the interface object that the application uses to perform the port predicate. Finally, we show that this agent is indeed an implementation of $ssmcast$.

Building the clone

Figuer 5.13 shows the `getClone()` method of the `MJRandomTreeMcastAgent`. Recall that when the application creates an agent object to pass to the `initSession()` method it does not have a session identifier. At the time the `getClone()` method is called, the agent has a session identifier. In this case it passes that session identifier to a one argument constructor to create a clone that knows how to contact the session leader. This constructor also sets a flag in this clone to indicate that it is not a session leader. The agent has the flexibility to create a clone that includes any arbitrary information. In this example the clone is of the same class as the session leader, but the clone can be an instance of another class that extends `MJAgent`. The agent can refresh this clone object any time during its life time by using the `refreshClone()` method. This would be used if the agent wishes to include more up-to-date information in the cloned object registered with the manager. In one of the multicast implementations we use the `refreshClone()` method to register a clone with up-to-date information of the child nodes included in its multicast tree. This clone is refreshed each time a child node leaves or joins the tree.

Initialization before the event loop

The `initProcessing()` method of the `MJRandomTreeMcastAgent` is shown in Figure 5.14. This code is common to the session leader and non session-leaders. There is a flag in the agent to indicate whether it is the session leader or not. All the agents insert

```
public void initProcessing() {
  routeAdd(new MJRouteRule().setType(ACCEPT).addOut(myLinkId()));
  routeAdd(new MJRouteRule().setType(INVITATION).addOut(myLinkId()));
  routeAdd(new MJRouteRule().setType(INVITE).addOut(myLinkId()));

  if (leader)
    routeAdd(new MJRouteRule().setType(JOIN).addOut(myLinkId()));
  else
    routeAdd(new MJRouteRule().setType(MCAST).addOut(appLinkId()));

  routeUpdate();

  if(!leader)
    send(new MJCommandCreateExternalLink(leader_adress));

  sendOnLink(appLinkId(), -1, -1, -1,
       new MJRandomTreeMcastInterface(myNetId(), leader));
}
```

Figure 5.14: The `initProcessing()` method of the `MJRandomTreeMcastAgent`.

routing rules requesting the router to forward messages of the types `JOIN`, `INVITATION`, and `INVITE` to itself. In addition to these routing rules the session leader (`leader`) also inserts a rule to forward `JOIN` messages to itself and non session leaders request the router to forward `MCAST` messages to the application. These types refer to the `Type` field in the `MJData` messages. The routing rules given to the `routeAdd()` method in Figure 5.14 are self explanatory. These routing rules apply only to `MJData*` (Figure 5.12) messages. These routing rules are based on the message type only. It is possible to add rules that take into account of all the header fields of the `MJData` messages. The `MJRouteRule` class also provides methods to set the rule to match ranges of the header fields and also to match these fields after applying a bit mask. All the routing rules are applied to a shadow copy of the routing table which is kept on the agent. The `routeUpdate()` command sends the updates to the router to update the routing table in the router. Once the routes are updated the agents that are not the session leader send a command to the router to

create an external link to the session leader. A reply to this request comes as a message and the agent can get this message once it goes into the event loop. The final action taken in the `initSession()` method is to send an `MJInterface` object to the application. The agent uses the `sendOnLink()` method to request the router to simply forward this message over a link that connects the application to the router. The `sendOnLink()` method is used to request the router to send a message over a link without consulting the routing table. This method is used to send message types that need to be sent only once.

Handling router replies

Recall that in the `initProcessing()` method (Figure 5.14) the agent issued a command to the router to get an external link to the session leader. On arrival of the reply to this command the `processExternalLink()` method (Figure 5.15) is invoked by the event loop. The agent implements this method to handle the reply. The reply from the router contains the link identifier of the new link (for simplicity, the error handling is not shown). The agent inserts a routing rule to send the `JOIN` messages to the session leader, then updates the routes and sends the join request. There is a possibility that the agent may not get an invitation from the current members of the tree. This can happen if all the leaf nodes die after receiving the `INVITE` message but before sending the `INVITATION` message and all the non-leaf nodes are full and therefore did not send `INVITATION` messages themselves. Therefore, the new node waits for a timeout and sends the `JOIN` request again. The agent sends an `MJCommandRemindMe()` message to the router instructing the router to send a message to wake up the agent to send the `JOIN` request again. With an `MJCommandRemindMe` message any object can be sent and the router sends back this object at the requested time. In this example the agent does not make full use of this feature and the only sends its *incarnation* number.

On receipt of a reminder message the event loop invokes the `processReminder()` method. In this method the agent checks whether it still does not have a parent node and checks whether the current incarnation number is equal to the incarnation number sent back

```
public void processExternalLink(MJReplyExternalLink msg){
 . . .
 if(!joined)
 {
   Integer leaderLinkId = msg.getLinkId();
   routeAdd(new MJRouteRule().setType(JOIN).addOut(leaderLinkId));
   routeUpdate();
   send(-1, -1, JOIN, getMyMJ());
   incarnation++;
   send(new MJCommandRemindMe(System.currentTimeMillis()+
      TIMEOUT, new Integer(incarnation)));
 }
 . . .
}
```

Figure 5.15: The `processExternalLink()` method.

```
public void processReminder(MJReplyReminder msg) {
  if(!joined && ((Integer)msg.getTask()).intValue()==incarnation)
  {
      send(-1, -1, JOIN, getMyMJ());
      incarnation++;
      send(new MJCommandRemindMe(System.currentTimeMillis()+
            TIMEOUT, new Integer(incarnation)));
  }
}
```

Figure 5.16: The `processReminder()` method.

by the router. If these conditions are true it again sends the `JOIN` request to the leader and a `MJCommandRemindMe` to the router in case this attempt to join also fails.

The methods that must be implemented to handle various other replies from the router are given in Figure 5.11. Their names are descriptive enough and we do not describe them further.

```
public void processData(MJDataLite msg) {
...
switch(msg.getType())
{
 case JOIN:
   ...
 case ACCEPT:
   numChildren++;
   children.add(new Integer(msg.getIn()));
   send(new MJCommandLinkInterest(msg.getIn()));
   routeAdd(new MJRouteRule().setType(INVITE).addOut(msg.getIn()));
   routeAdd(new MJRouteRule().setType(MCAST).addOut(msg.getIn()));
   routeUpdate();
   break;
 case INVITE:
   ...
 case INVITATION:
   if(!joined)
   {
   send(new MJDataLite(myNetId(),-1,-1,ACCEPT, new String("Hello")));
   joined = true;
   parentLink=msg.getIn();
   send(new MJCommandLinkInterest(msg.getIn()));
   }
   ...
}
}
```

Figure 5.17: The `processData()` method.

Processing overlay-specific messages

The `processData()` method of the `MJRandomTreeMcastAgent` is shown in Figure 5.17. The agent uses `MJData` messages to both implement the tree construction protocol and send application data on the overlay. Recall that the `initProcess()` method of the agent inserts routing rules for different types of `MJData` messages for its network. These rules guide some `MJData` messages to the agent and the agent handles these messages in the `processData()` method. For example in the code listing in Figure 5.17 on

receipt of an `ACCEPT` message the agent increments its count of the children and inserts two routing rules into the routing table. These routing rules instruct the router to forward both `MCAST` and `INVITE` messages to the new node. When the router gets these messages from upstream nodes it forwards these messages to the new node without any intervention from the agent. The agent also tells the router that it is interested in the link to the new node by sending the command `MJCommandLinkInterest`. This command requires the router to monitor this link and inform the agent if it goes down. The agent must implement the `processLinkDown()` method to handle such a situation. The handling of the `INVITATION` message in Figure 5.17 is self explanatory.

Interface object

Figure 5.18 shows the code for the `MJRandomTreeMcastInterface`. The agent extends `MJInterface` according to its needs. The application uses this interface object to send and receive messages. In this case the interface object also enforces the port predicate to some extent by not allowing non-source nodes to send messages and not allowing source node to receive messages. This is done by throwing an exception if the sender tries to receive messages or a receiver tries to send messages. Error handling in MayaJala is discussed in Section 5.3.5.

The communication type of `MJRandomTreeMcastAgent`

When we introduced `MJRandomTreeMcastAgent` we mentioned that it is an implementation of $ssmcast$. Now we show that this is indeed the case.

Let all the applications that communicate over a given session that uses this agent be in the set P. Let all the send-enabled applications in P be in S and all the receive-enabled applications be in R. The messages sent into the router by an application p is denoted by a sequence $M_s(p)$ and the messages delivered to the application by the router is denoted by $M_r(p)$.

The interface object (Figure 5.18) returned to the application on the session leader

```
public class MJRandomTreeMcastInterface extends MJInterface {
  boolean source;
  public MJRandomTreeMcastInterface(MJNetId nid, boolean src) {
    super(nid);
    source = src;
  }

  public void send(Object o) {
    if(!source)
      throw new MJLocalProtocolException(...);
    MJPacket p = new MJPacket();
    p.sendObject(new MJDataLite(myNetId, -1, -1,
      MJRandomTreeMcastAgent.MCAST, o), sessionLink.getSink());
  }

  public Object recv() {
    if(source)
      throw new MJLocalProtocolException(...);
    MJPacket p = new MJPacket();
    return ((MJDataLite)p.getObject(sessionLink.getSource())).getData();
  }
}
```

Figure 5.18: The `MJRandomTreeMcastInterface` class.

is send enabled but not receive enabled and the agent makes sure that the interface objects returned to non session-leaders are only receive enabled. It is therefore clear that $(|S| = 1) \wedge (R \cap S = \emptyset)$ at any given time after the start of the session. This is the port predicate of $ssmcast$. In this particular case the agent enforces the port predicate rather than expecting the applications to follow the port predicate collectively; the `MJRandomTreeMcastInterface` shown in Figure 5.18 throws a protocol exception if a send-enabled process tries to receive messages and vice versa.

The protocol that we described in the beginning of the Section 5.3.3 ensures that an agent only joins a single parent. Each agent inserts routing rules into the router to forward `MCAST` messages from its parent to all its children and also to the application in its own

node. This protocol builds a delivery tree, rooted at the single source, for `MCAST` messages and this ensures that there is only a single path from the source application to any other application in the same session. The links and the routers do not reorder messages. If we look at the messages sent and received when the delivery tree is stable it is clear that the messages delivered to all the applications are subsequences of the messages sent by the sender.

If a node is disconnected from its parent it uses the same protocol to join the session again by sending a `JOIN` request through the source, which is also the root of the tree. Suppose that a node p_i loses its parent after receiving a message x from the source. Since p_i sends a new `JOIN` request through the root the message that the root sends down the tree instructing the nodes to send invitations to p_i will be received after the message x at all the nodes. The new parent only adds a rule to the router to forward `MCAST` messages to p_i only after p_i accepts its invitation. This ensures that only the messages sent by the source after x will be received at the new node. Therefore even after re-joining the tree it is guaranteed that a node only receives a subsequence of messages sent by the sender. The port predicate ensures that there is only one such sender. It is clear that $\exists_{q \in P} \forall_{p \in P} [M_r(p) \sqsubseteq M_s(q)]$ at all times during the session and this is the message predicate of $ssmcast$. Therefore, we conclude that `MJRandomTreeMcastAgent` implements $ssmcast$.

5.3.4 Layering agents

The agents also can use communication types to implement other communication types. An agent for $atoa$ (all-to-all multicast), named `MJAlltoAllAgent`, was implemented to illustrate this. `MJAlltoAllAgent` uses $gather$ and $ssmcast$ (single source multicast) to implement $atoa$. On the session leader `MJAlltoAllAgent` starts a $gather$ session and an $ssmcast$ session and gets session identifiers for these two sessions. This agent also gets a session identifier for its own session—this is the session identifier that other nodes see. Other nodes join the session and their agents get two session identifiers from the session leader to join the $ssmcast$ session and $gather$ session. Messages from the application

```
...
if (leader) {

initSession(new MJRandomTreeMcastAgent(),
                                        new MJSessionCallBack(){...});

initSession(new MJTrivialGatherAgent(), new MJSessionCallBack(){
  public void onReady() { joindata[1]=iface.getNetId(); }

  public void onRecv(Object data) {
    if (mcast != null)
    {
      mcast.send(data);
      send(0, -1, TO_APP, data);
    }
  }
});
}
...
```

Figure 5.19: A section of the `initProcessing()` method of the `MJAlltoAllAgent`.

are routed to the agents and the agents send the messages along the $gather$ session to the session leader and the session leader sends the message again to all the participants over the $ssmcast$ session. The session leader also must send a copy to its own application.

Agents have a slightly different interface to start and join a session. A section of the `initProcessing()` method for the `MJAlltoAllAgent` is shown in Figure 5.19 to illustrate this. The agents do not have a reference to the MayaJala instance and the `MJAgent` superclass provides a similar interface. The `initSession()` method for agents registers a callback object with the `MJAgent`'s event loop. This is shown in Figure 5.19 for the `initSession()` call for `MJTrivialGatherAgent`. The agent's event loop has a mechanism for receiving messages on these sessions without blocking and an agent can take advantage of this without creating its own threads. The `joinSession()` method also registers a similar callback.

While designing agents in this manner is possible, it is not the recommended way to implement agents. We assume that the type programmers build the communication types by building their own overlay networks taking full advantage of the facilities provided by the MayaJala router. Because of this reason we do not explore layering any further.

5.3.5 Error reporting and enforcing the port predicate

There are two main classes of exceptions in MayaJala.

1. Protocol exceptions (`MJProtocolException`).
2. Network exceptions (`MJNetworkException`).

These are again divided into two types: local and remote. The protocol exceptions occur when an application violates the port predicate. This error is reported to the offending application as a local protocol exception (`MJLocalProtocolException`). This was illustrated in the `MJRandomTreeMcastInterface` (Figure 5.18). Such an exception can be reported to other nodes as an `MJRemoteProtocolException`. Network exceptions occur if the implementation cannot follow the message predicate because of a network failure. In MayaJala all the exceptions are implemented as subclasses of Java's `RuntimeException`.

The error handling mechanism in this version of MayaJala is very rudimentary. Only one of our implemented agents reports even local protocol exceptions. The $ssmcast$ agent that we used as an example in this discussion throws exceptions if a node violates the port predicate. Also it makes sure that the session leader is send enabled and all the other nodes are only receive enabled. It is not clear whether it is meaningful to report some exceptions to the processes other than the process that violated the port predicate. For example in $ssmcast$ if a receiver violates the port predicate by attempting to send a message an implementation can simply prevent sending and report the local error to the offending process. Other than for debugging purposes the other processes can do without getting a report of this violation since the session can continue without the offending process. Similarly

if the source process in an $ssmcast$ session is disconnected from the rest of the processes it is not possible to detect whether this is a transient network failure or a violation of the port predicate. It may also be very expensive to check for the port predicate in some cases such as ensuring that there are as many receivers as the largest N of senders in $manycast$. It may be possible to deliver messages from some of the senders to the required number of receivers. In this case it is not clear whether to report the port predicate violation to all the nodes or just to the senders that have N larger than the available number of receivers.

Error handling and reporting is not a trivial issue. Even MPI makes the users responsible for the correct use of the communication primitives and does not require an implementation to check whether a collective communication primitive is used correctly across all the participants. The error handling issue is not addressed any further in this thesis.

5.4 The complete architecture

The Figure 5.20 shows the complete architecture of MayaJala. The difference between this view and the type programmers' view is that the type programmers assume that there is only one agent running on the MayaJala instance. Type programmers also do not see the internal workings of the router, the manager, and the link monitor. The router and the manager provide a message-based interface to the agent and the agent interacts with them using this well defined interface. The link monitor is a private module of the router.

As the developers of the middleware we only build the router, the manager, and the link monitor; as explained later, even the link monitor can be expected to be built by third parties. We envisage that the agents will be developed by different programmers. That is a main premise of this thesis; the type programmers can explore the design space to implement communication types and applications can use them transparently. In such a scenario it is important that clear execution boundaries be established between these agents. In this prototype implementation we do not explore the issues of providing a completely secure environment in which agents can be deployed. However, we attempt to provide reasonable protection to agents from each other and also protection to the router, manager,

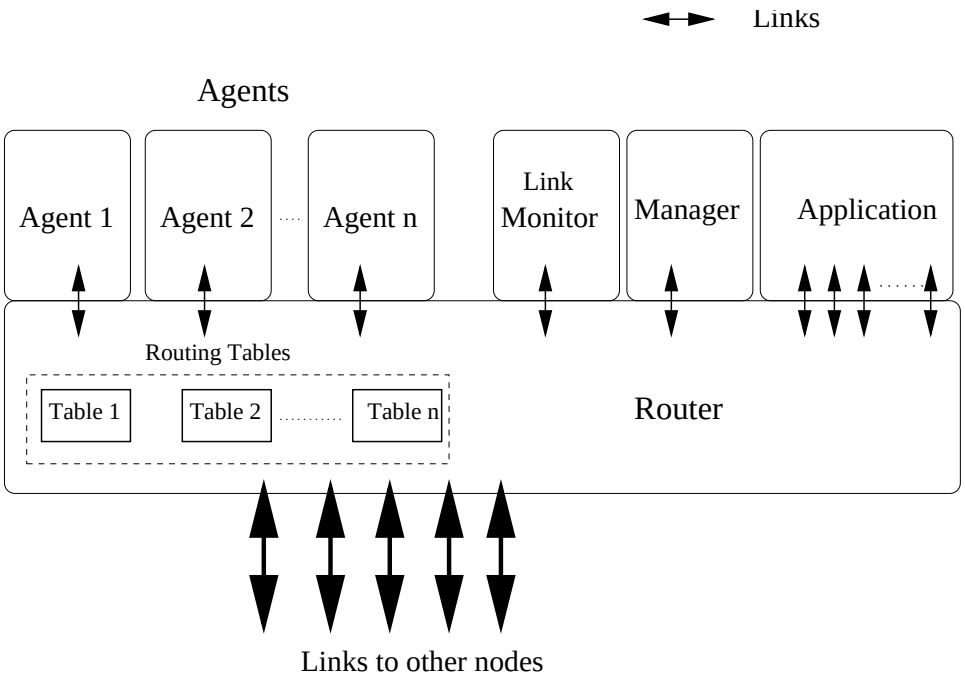

Figure 5.20: The architecture of MayaJala.

and the application. We assume that the agents are not malicious, but the correctness of agents is not assumed. It is also important that the agents can be written without knowing the details of their final deployment environment and they should be able to take advantage of the deployed environment. The architecture of MayaJala is driven by these concerns.

5.4.1 The router

The MayaJala architecture is based on the assumption that communication types can be implemented using overlay networks. A basic functionality required by an overlay network (for that matter any network) is message routing. The router provides this common functionality to all the agents running on the MayaJala instance.

All the components of MayaJala are connected to the router. The router also has external links that connect it to other MayaJala instances. The router contains a routing table for each agent. These routing tables contain rules that the router must follow to forward and replicate messages on each agent's network.

The router is implemented as a single threaded event driven program. It responds to messages coming along the links by either forwarding them on some set of outgoing links or taking some other actions. Despite its importance in the MayaJala architecture, the router is a very thin, light-weight module. Its main services to agents are forwarding messages, providing a timer service, and monitoring links; the bulk of the last service is offloaded to the link monitoring module transparent to the agents.

The operation of the router can be described with respect to the links and messages.

Links

There are two types of links in the router.

1. External links.
2. Internal links.

External links hide the operating system abstractions such as TCP, and UDP from the agents and provide a message oriented stream abstraction to the agents. Links do not guarantee that all the messages are delivered. To handle the possible mismatch of the speed between the incoming and outgoing links there is a queue associated with each link. The router may drop messages from these links if the queue length reaches a certain maximum size. However, before dropping messages the router informs the agent responsible for the offending network. This prototype of MayaJala does not enforce message dropping. However, such a mechanism is required to make different MayaJala instances loosely coupled [55].

This prototype implementation uses TCP sockets to implement external links. We took this decision in favor of fast prototyping. TCP itself provides flow control, congestion control, and packet ordering. If we had opted for UDP instead of TCP we would have to implement these to provide an ordered message oriented stream to agents; this is tantamount to reinventing TCP. While the primary reason for selecting TCP for the prototype was fast prototyping, we note that there are several end-system multicast implementations that uses

TCP, such as Overcast [47] and ALMI [75]. Also, we note that Krasic et al. [53] have argued in support of TCP for even streaming multimedia applications. Since the router hides the implementation of links from the agents, it is simple to change this decision to use TCP. For example, the router can transparently use Stream Control Transmission Protocol (SCTP) [86], which itself provides message oriented stream abstraction, to implement the external links. SCTP is not available in all platforms and MayaJala instances could negotiate to use SCTP where available to build the links. Agents do not have to know about this and depending on its deployed platform the agents get to use new implementations of the links.

The current implementation only creates a single link between any two MayaJala instances. This link is multiplexed between all the agents. However, the design allows for creation of multiple links as the need arises transparent to the agents. The design also makes it possible for the router to transparently use native multicast facilities, if available, to send copies of messages to multiple destinations.

The router collects the statistics on the links and uses the link monitor (Section 5.4.3) to do extensive measurements on the links. The router itself detects lost connections and informs the agents who have registered interest in the failed links.

The single thread of the router never blocks on any task (except the `select()` call). The internal links help the router to hand over messages to agents, the manager, and the application without blocking. An alternative would have been to use an upcall. However, an upcall into the application and agents may block indefinitely (or it may not return at all), thus blocking the event handling thread. The internal links are similar to the *queues* described by Welsh et al. [94] to introduce explicit control boundaries between threads.

Internal links are implemented using Java NIO `Pipe`s. The `SourceChannel` and `SinkChannel` associated with a `Pipe` are `selectable` and the same select mechanism can be used in the router for pipes as well as for sockets. We implemented a wrapper class, `MJPipeLink`, around the pipes so that when sending and receiving large messages only the header object is sent on the actual selectable channel, but the payload is transfered over

shared memory. There is a shared space private for each `MJPipeChannel`. However, this is completely transparent to the components that use the `MJPipeLink`. Also, the payload of the large messages are serialized inside the `MJPipeLink` and users do not get a reference to the serialized object, thus eliminating the danger of users corrupting this shared space.

Messages

The router must respond to three types of messages.

1. `MJCommand`.
2. `MJRequest` and `MJResponse`.
3. `MJData`.

Agents use `MJCommand` messages to get services from the router. For example, in Figure 5.17 the agent uses `MJCommandLinkInterest` to inform the router that it wants to be notified if a particular link goes down. The router adds the agent's link identifier to a table in that particular link. Note that the router identifies different entities running on it by their link identifiers. If that particular link goes down the router informs all the interested parties. This is only an example of a command message and there are commands to open and close links, request link statistics, send messages bypassing the routing table, start a timer, and update a routing table.

The `MJRequest` and `MJResponse` messages are for two routers to communicate with each other. In the current prototype these messages are used on two occasions. One occasion is to set up a link between two routers. To make sure that there is only single TCP connection between two routers there is a handshake protocol and the router uses `MJRequest` and `MJResponse` messages to implement this protocol. The second occasion is when closing down a TCP link. The routers do not shutdown a link if there is any party interested in the link on either end. There is a protocol to ensure this and the routers

use MJRequest and MJResponse messages for this. Both of these actions are triggered by MJCommand messages.

The structure of an MJData message is shown in Figure 5.12. The router provides a general purpose message forwarding mechanism for MJData messages. It uses several routing tables to forward packets. Each packet belongs to a single network and carries a network identifier (MJNetId) that uniquely identifies a network; there is a network for each session of a communication type and MJNetId also uniquely identifies the session. The router uses the MJNetId as an index to the routing table. The routing table contains a list of rules keyed by the incoming link, source, destination, and the type of the message. This type field has a meaning only within the network that the message belongs to and is not to be confused with the message types (MJData, MJCommand, MJRequest) as the router sees them. The router matches the incoming message against all the rules in the routing table to get a list of out going links and forwards the message along those links. Apart from this general routing facility, the router also provides different classes of routing rules. The current implementation provides one additional routing class that forwards messages to a random link out of the set of links indicated in the routing rule. It is possible to extend the routing classes easily to other link selection functionality, such as selecting the least loaded link. MJData messages can be matched against the rules directly or through a set of optional bit masks defined for each field to be matched. Routing rules can also be written to match ranges of these fields. As shown in Figure 5.20 these links can be the links to other MayaJala instances in other nodes or links to the other components of the MayaJala instance such as the application, the agents, and the manager. All communication between the local components of a given instance of MayaJala as well as communication between MayaJala instances goes through the router.

On receipt of an MJData message, the router sets its In_link field to the link identifier of the link that the message was received on. Then the TTL field is decremented before applying the routing rules to the message header. If TTL is zero the message is dropped and the router informs the agent responsible for the overlay network in the local

MayaJala instance.

The three fields, `MJNetId`, `TTL`, and the `In_link`, are the only fields that have semantics defined by the router. The other fields do not have any predefined semantics attached to them. It is up to the agents to use these header fields and the payload for their own use. Even though the three fields `Source`, `Destination`, and `Type` have meaningful names, these meanings are not enforced. These field names are in the header as a convenience for writing agents, since we believe that an overlay may need such identifiers. However, they can be treated as a single 12 byte value by the agents.

5.4.2 The manager

The manager is responsible for starting agents. It also acts as the repository for the agent clones and hence is a resolver for the session identifiers. Most of the manager's functionality was discussed in Section 5.3.1 where the life cycle of an agent was presented.

Managers must communicate with each other to resolve the session identifiers and to download the agent. The manager is implemented as an agent but with more privileges[3]. These extra privileges allow the manager to create internal links. An agent cannot create internal links. Once an `MJPipeLink` is created a reference to it must be registered with the router and this requires a reference to the router (`MJPipeLink` cannot be serialized and hence cannot be sent to the router in a message). The manager has a reference to the router and agents do not have such a reference. Also note that the manager can start the agents by loading them with different class loaders with restricted security policies.

The manager maintains its own routing table in the router and also uses the generic `MJData` message type for communication on its own network. Given a MayaJala instance the manager's session identifier can be inferred from that. Managers use their own networks to communicate with each other to resolve session identifiers and download agents.

Agents also send messages over the manager's network to get services from the managers. In the current implementation such communication only occurs when an agent

[3]This prototype does not implement the manager as a subclass of `MJAgent`. However, its operation is similar to an agent.

registers its clone with the manager or refreshes it.

The manager is also implemented as an event driven program. When initializing agents the manager calls some methods defined in the agents. To make sure that none of these methods take an arbitrarily long time (or block), the methods that the manager invokes are defined in the super class (`MJAgent`) as `final` methods.

5.4.3 Link monitor

Measuring the bandwidth and latency of Internet paths is a specialized task. Bandwidth measurement is an ongoing research area [56, 89]. It is specially challenging to measure the bandwidth in a non-intrusive manner. Also, even a seemingly simple task like measuring the latency between two points is more involved than one might think. Simple techniques such as round trip time measurements do not reflect the asymmetry in the Internet paths and hence the asymmetry in the latency. There are different techniques to address the link measurement problem and we expect that these techniques would get more sophisticated over time. It is important that the link monitoring functionality in MayaJala be isolated in a separate module. This allows easy replacement of one implementation with another that uses more sophisticated techniques.

Some overlay networks also require the knowledge of the underlying network topology (over the Internet). Nakao et al. [70] argue that probing the network by each overlay is an untenable strategy. They propose a shared routing *underlay*, which discovers the topology and individual overlays query this shared layer. In MayaJala architecture the link monitoring module is a good place to implement such a common topology discovery mechanism.

The link monitor implements the link measurement techniques. In the MayaJala architecture the link monitor is implemented as a module similar to agents so that it is a simple task to replace one implementation with another. The current prototype version of MayaJala does not implement this module. As mentioned above, link monitoring is a specialized task and we did not venture into this area. One advantage of this design is that

agents can take advantage of new link monitoring techniques at deployment time. Agents simply use the command message provided by the router to get the link measurements without being aware of the module that does the measurement.

5.5 Related work

The architecture of MayaJala is influenced by the staged event driven architecture of SEDA [93]. SEDA is proposed as an architecture for scalable Internet services. In SEDA, stages are self contained application components that have an event loop and are connected to other components by event queues. In MayaJala the router, agents, the manager, the link monitor, and the application are all independent components connected to the router via message links. Except for the manager and the router all the other components of MayaJala are implemented by third parties. We do not have control over them. It is important that these components run on their independent threads and the router has to avoid any upcall mechanism to prevent blocking. In addition to being a proven scalable architecture for Internet services, we believe that a SEDA-like architecture is appropriate for MayaJala to accommodate third party components; agents, arbitrary applications, and link monitors.

MPI provides several multiparty communication types, collective communication routines in MPI parlance, to parallel applications. We investigated a public domain implementation of MPI called LAM/MPI [58] to find out how these collective communication routines are implemented. The version that we studied was LAM/MPI 6.1. LAM provides a library that implements the MPI routines and a runtime environment to launch applications. LAM only has to provide support for the different implementations of a fixed set of communication primitives — providing support for unknown set of communication primitives is not an issue. MPI requires applications to be “safe”; in a safe application communication can progress irrespective of whether the communication calls progresses asynchronously or not. LAM is single threaded and depends on the safeness of the application to progress the messages. All MPI collective communication routines are blocking and they are truly collective communication calls—that is, all the processes in the communicator must call the

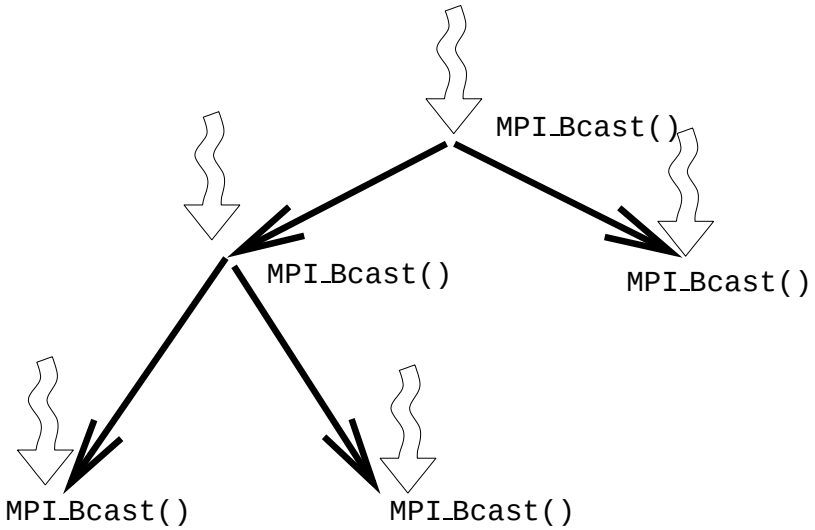

Figure 5.21: The message forwarding by processes in `MPI_Bcast()`. All processes must arrive at the `MPI_Bcast()` call without blocking in a safe application.

routine in unison. LAM gets the processes in the communicator to form a delivery tree to implement collective calls such as `MPI_Bcast()`. This depends on each process that receives the message to forward it down the tree. In a safe application all the processes in the `communicator` will call the routine without blocking in a call before that (Figure 5.21). This helps a single threaded library like LAM/MPI to progress messages. It is interesting to note that the return from a call to most MPI collective communication routines does not mean that the message has been received by the receiver—it does not even mean that the message has been sent out. It only means that the message buffer can now be reused. Therefore, in LAM/MPI a call to any MPI routine may end up progressing messages left behind by earlier calls. Note that the LAM architecture has a daemon process which can be used to progress messages in the *lamd* mode. However, even in this mode the daemons do not know about the communicator and hence the delivery tree. The processes must participate in progressing the messages along the tree even in this mode.

LAM can get away with such a single threaded model with the help of the safety property and also because the processes in a communicator do not change over the time. MPI communicators are static. The multiparty communication types that we are interested

in supporting are not confined by such a rigid requirements as MPI. In MPI the idea of collective communication is either directly or indirectly synchronization. On the other hand in the distributed computing world the idea of having a communication paradigm is exactly the opposite—for example, multicast separates the senders from the receivers. This level of indirection allows the set of senders and receivers to grow and shrink without informing the current set. However, this requires asynchronous handling of the joins and leaves and also the message progress.

Macedon [82] is a domain specific language to build overlay networks. Protocol states and transitions are described using the Macedon language. This overlay description can be used to generate either C++ code to run on a real system or ns-2 code for simulation. Macedon provides protocol layering. Upper layers register upcalls with the lower layers to handle messages. As we mentioned previously, such an upcall/downcall mechanism is not suitable for MayaJala. The Macedon protocol stacks are library based protocols stacks; that is one protocol stack is isolated from another and there is no sharing of a lower layer by two or more upper layers. In other words there is no run time system like the MayaJala router that allows sharing the resources. Macedon also does not provide any deployment mechanism for the overlay. It is interesting to note that Macedon overlays provide *multicast*, *anycast*, and *collect* primitives to the upper layers to send and receive messages on the overlay. In contrast, the MayaJala approach is to provide facilities to build different communication types and not a fixed set of communication primitives. Furthermore, building overlays in MayaJala is simply a means of implementing communication types and we hide the fact that overlays are being used from the applications. However, it is possible to use a Macedon like domain specific language to write agents to run on the MayaJala router; after all agents are state machines that change states and take actions depending on the messages. This is an interesting avenue for future research.

The Spread [2] system implements group communication protocols over wide area networks. It is built on the *extended virtual synchrony* model of group communication. In Spread there are long running daemons that set up the message delivery network. This

network provides the membership and message ordering services. An application connects to a daemon to send and receive messages. Joins and leaves are handled by the daemon. Each daemon contains a routing module and it computes the message delivery tree; for each *site* a separate tree is constructed rooted at that site. There are two link level protocols that connect daemons; the *Hop* protocol connects two daemons over a point-to-point link and the *Ring* protocol connect two daemons over a multicast network, such as a LAN. In contrast to MayaJala, Spread provides a fixed set of protocols (multicast family) for group communication.

The agent deployment mechanism of MayaJala is reminiscent of the techniques used to disseminate protocols in active network toolkits like Ants [95]. However, in MayaJala the agent deployment is a one time action rather than a general packet processing method (in this prototype version there is no dynamic code loading at all and only the data to instantiate the agent object are downloaded). The MayaJala router simply forwards messages and does not do any processing. The messages are not active either; they do not carry code or an index into code to process themselves. However, it is interesting to note that some multiparty communication paradigms have been implemented using active networks; the multicast implementations presented in [95] and Concast [19] are examples.

There is a large body of work that implements overlay networks using distributed hash tables (DHT) [79, 88, 84, 98]. Several multiparty communication paradigms, such as multicast [80, 99], anycast [23], and manycast [23] have been implemented on DHT based networks. While it is possible to build DHTs in MayaJala, we have not explored this avenue. In the MayaJala model, only the nodes that are in the communication session participate in the overlay while in DHT based multicast implementations the multicast trees are built on a bigger *universal* DHT. One of the criticisms leveled against DHTs is the lack of such a universal DHT [50]. Note that there is an ongoing effort [72] to deploy a public DHT within the PlanetLab [77] network using Bamboo DHT [81] to address this problem. If such a universal DHT is available one interesting possibility is to use it as a network that sits below the MayaJala router (as the IP network does in the current version) that provides

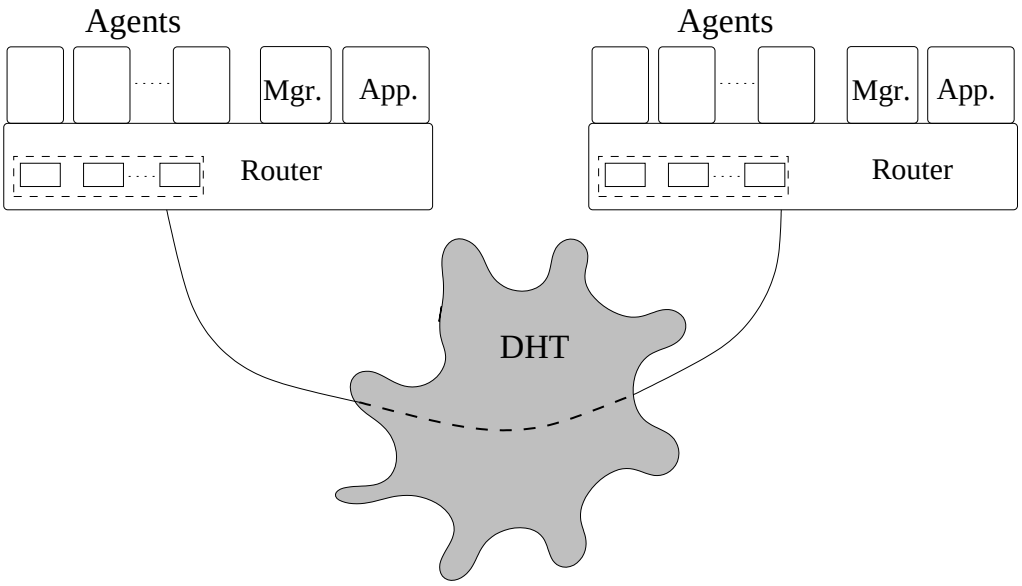

Figure 5.22: The MayJala-links over a DHT.

richer communication facilities. As shown in Figure 5.22 the MayaJala router can create external links between MayaJala instances over the DHT or simply use the DHT network to locate endpoints. In such a scenario MayaJala instances can be identified by a naming system not tied to the IP addresses.

The Click [68] modular router is a configurable router. Click allows different packet processing elements to be assembled to process packets. In contrast, the router in MayaJala provides a fixed set of basic functionality to all the agents. We expect this functionality to grow as we get more experience in implementing different agents, but MayaJala does not provide functionality to extend the router as agents wish; agents are implemented by third parties and we do not want to install arbitrary functionality in the router that can affect other agents. The routers in all MayaJala instances should provide the same functionality so that agents can be written assuming this functionality. Therefore flexibility for extensibility is not a central issue. However, the architecture of the router is very modular and we expect that this will help future extensions.

x-kernel [46], Horus and Ensemble [14], Click [68] modular router, and Scout [69]

all have the idea of building protocols as layers of simple micro protocols. Even though MayaJala supports building communication types using other communication types, we expect type programmers to implement communication types as monolithic units. We believe that communication types have not been studied enough to come up with a good layering scheme that fits all communication types. Only long term experience in building and using communication types can provide such an insight. MayaJala is a good platform to build communication types to gain this insight.

5.6 Contributions

We designed MayaJala to address the third issue discussed in Chapter 3; to provide a system that facilitates the implementation and deployment of communication types. The prototype demonstrates that communication types can be implemented independent of the applications as loosely coupled modules, agents, and that they can be deployed transparently to the applications that use them. A communication type provides a clear contract between application programmers and type programmers. This helps the modularity of the agents. The fact that different implementations can be used for different sessions without any code change to the applications shows that Agents are loosely coupled to the applications. MayaJala also provides support to simplify the development of agents. We also demonstrated the code level simplicity of the applications that use MayaJala and communication types. Apart from highlighting these benefits the prototype also fulfilled a role necessary for the presentation of the framework as discussed below.

The communication type system provided the conceptual framework for the design of MayaJala. From the model it is not obvious how the ideas developed from the model can be used in practice. As we noted before there is no representation of communication types in the program text of applications. The communication type system provides a role similar to design patterns. The descriptions of design patterns accompany examples and these examples help to map the abstract idea of the design pattern to the real world situations. Similarly, our prototype and example applications are integral parts of the framework that

demonstrate how communication types and associated ideas are mapped to the real world applications.

Chapter 6

Evaluation

We evaluate our implementation of MayaJala and the idea of communication types on several dimensions.

- **Simplicity of the application code**: We demonstrate that communication types allow programmers to write simpler programs, using an example application that we implemented. We use the same application to demonstrate that one implementation of a communication type can be transparently switched with another without changing the number of lines of communication related code.

- **Support for building agents**: The second aspect that we evaluate is the support provided by the middleware for implementing communication types. We evaluate this by implementing two multicast algorithms and comparing them with equivalent overlay networks implemented in Macedon.

- **Deployability**: We identify and list the support required from entities such as firewalls and operating systems to deploy a MayaJala session. We show that a MayaJala session can be deployed with minimum support from these entities. We also show that an implementation of a communication type or the communication type itself can be selected and deployed without any coordination from the session members.

	Chat Sever	Chat Client
Communication related LOC	7	7
No. of changes to switch an implementation	1	0
No. of changes to switch a type	1	0

Table 6.1: Complexity of the code.

- **Overhead interposed by MayaJala middleware**: Finally, we evaluate the cost interposed by the middleware itself and we show that this is negligible in comparison to wide area Internet costs.

6.1 Simplicity of the application code

To demonstrate the simplicity of the communication code in applications that use MayaJala, we implemented a more complete application than the example applications that we presented in Chapter 5. This application constitutes a chat server and chat clients. The chat server acts as moderator for the conversation carried out by the chat clients. Messages from each client are collected at the chat server. The server "sanitizes" the messages and sends each one to all the chat clients including the originator of the message. There is only one common chat session per server. This is the application that we discussed in Chapter 1 and now we present the actual implementation of that application that uses communication types.

The code for the chat server is given in Figure 6.1 and the code listing for the client is given in Figure 6.2. The client is an application with a graphical user interface (GUI) and we removed most of the the GUI code from the client code.

This application uses two different communication types, $ssmcast$ and $gather$. The chat server starts two sessions, one for $ssmcast$ and one for $gather$. Note that the the port predicate of $gather$ requires that there is always one receiver and this requires the session leader to be the receiver and to be alive for the whole duration of the session. The port predicate of $ssmcast$ requires a single sender at all times and the server takes that role too.

```
public class Moderator {
 public static void main(String[] args) {
   MayaJala mjinstance = new MayaJala();
   MJInterface gather=
         mjinstance.initSession( new MJTrivialGatherAgent());
   MJInterface mcast=
         mjinstance.initSession(new MJRandomTreeMcastAgent());
   gather.recvEnable();
   mcast.sendEnable();
   ...
   while(true)
   {
     String msg = (String)gather.recv();
     mcast.send(sanitize(msg));
   }
 }

 public static String sanitize(String str)
 {
   ...
 }
}
```

Figure 6.1: The chat server.

The server receives messages coming along the $gather$ session, processes the messages, and sends out the processed messages along the $ssmcast$ session. Even though the chat server has a fairly complex communication pattern, there are only 7 communication related lines of code in the server. The number of lines of code in the user application does not depend on the complexity of the implementation of the communication type. The implementation of any one of the communication types can be selected just by modifying a single line of code. The server only has to give an agent of a different implementation to change the implementation. We successfully tested this application with two different implementations of $ssmcast$, `MJRandomTreeMcastAgent` and `MJTestAgent`. Similarly, to change the communication type with a conforming communication type only one line has

```
public class Client extends JFrame . . . {
 . . .
 public Client(MJInterface gather, . . . )
 {
   . . .
   send = new JButton(”Send”);
   send.addActionListener(this);
   . . .
 }
 public void actionPerformed(ActionEvent e)
 {
   if(e.getActionCommand().equals(”Send”))
   {
     . . .
     gather.send(name+” : ”+input.getText());
     . . .
   }
   . . .
 }
 . . .
 public void showDiscussion()
 {
   . . .
   SwingUtilities.invokeLater(new TextSetter((String)mcast.recv()));
   . . .
 }
}
 public static void main(String[] args) {
   MJInstance mjinstance = new MayaJala();
   MJInterface gather = mjinstance.joinSession(new MJNetId(args[0]));
   MJInterface mcast = mjinstance.joinSession(new MJNetId(args[1]));
   gather.sendEnable();
   mcast.recvEnable();
   Client cl = new Client(gather, mcast, args[2]);
   . . .
 }
}
```

Figure 6.2: The chat client.

to be changed. Since $atoa$ conforms to $ssmcast$ we also successfully tested this application by switching the communication type $ssmcast$ with $atoa$ using the $atoa$ implementation `MJAllToAllMcastAgent`.

When the chat client starts, it joins the two sessions created by the server. The clients get the session identifiers of the $ssmcast$, and $gather$ sessions on the command line. The client is a GUI based application and the user gets a pane to input messages and the messages from all the other clients in the session are displayed in another pane. The majority of the code in this application is for the GUI. There are only 7 communication related lines of code. The change of the implementation of a communication type does not affect the number of lines of code and nor does it results in any change of code. The client is not even aware of the change. Similarly the client is not aware of the change of the communication type either and no change necessary in the client. This shows that agents are loosely coupled to the application. These results are summarized in Table 6.1.

These results are possible because of the modularity of the agents and the loose coupling between the application and the agents. The generic API facilitates the loose coupling, but as we mentioned in Chapter 4 the API by itself does not define the correct use of the API; the port predicate of the communication type defines the correct use of the API. Communication types also provide a clear division of the responsibilities of application programmers and type programmers and this also helps the modular design of agents. In addition, the facilities provided by MayaJala to implement and deploy agents contributed to the above benefits.

6.2 Support for building agents

Macedon [82] is considered to provide an efficient domain specific language to build overlay networks. The agents also build overlay networks. We compare two agents against similar overlay networks built using Macedon to evaluate MayaJala's support for building overlays—in other words agents.

We implemented the Overcast [47] overlay network using MayaJala. We com-

	MayaJala	Macedon
Overcast	289	311
Random Tree	82	142

Table 6.2: Number of lines of code.

pletely implemented the complex tree building, tree evolving, and maintenance algorithm of Overcast. We omitted only the data caching in Overcast and also in our implementation all the nodes are tree nodes—in Overcast there are nodes in the delivery tree and also nodes that just connect to the overlay to get data. We compared our implementation against the Macedon implementation of Overcast (in the Macedon version released on October 08, 2004) in terms of number of lines of code. We counted the number of semicolons not including: print statements, debug statements, commented code and comments, and imports lines (in our Java code) from both implementations. Our implementation spans several classes and several files and we included code in all these files (we also included a class that is not specific to the Overcast implementation). We only counted code in one Macedon file, `macedon.mac`, even though it uses filters defined in another file.

The results are given in Table 6.2. Our implementation consists of 289 lines of codes while the Macedon implementation has 311 lines of code. These numbers are very encouraging specially considering the fact that Macedon is a domain specific language for building overlay networks while our implementations use the generic and popular language Java. One of the reasons for the small amount of code in the MayaJala implementation is the functionality provided by the router to measure and monitor links. The Overcast algorithm requires a node to measure the bandwidth to its parent, grandparent, and siblings. The MayaJala implementation uses the measurement and monitoring functionality provided by the router, rather than attempting to measure the bandwidth itself.

We implemented Overcast especially to compare against the Macedon implementation. We also have an agent, `MJRandomTreeMcastAgent`, which builds a delivery tree somewhat similar to the Macedon `randtree`. This agent was not built just for the purpose of comparison, but due to the similarity of this agent to the Macedon `randtree`, we use

it for comparison as well.

Macedon `randtree` builds a single source delivery tree. Each node has a maximum number of children that it can have. If the node already has this number of children then it directs join requests, randomly, to one of its children. New nodes first send the join request to the root node. The agent, `MJRandomTreeMcastAgent`, also restricts the number of children that a node can have. A new node sends the join request to the root. The root sends the join request down the tree and if it does not have the maximum number of children, also sends an invitation to the new node. Nodes down the tree do the same. The new node simply selects the first invitation (random selection or any other criteria is possible) that it receives and joins the inviter. Since there is at least one leaf node in the tree, it is guaranteed that the new node gets an invitation. Note that just selecting a random node to forward the join request is simpler in MayaJala since the router itself provides that facility.

The number of lines of code in these implementations are given in Table 6.2. `MJRandomTreeMcastAgent.java` has 82 lines of code and `randtree.mac` has 142 lines of code. While the number of lines of code is not a definitive measure of the simplicity of the code, the significant difference between these two numbers shows that, even after discounting the effect of different coding styles and different functionalities, MayaJala's support for building overlay networks is at least as good as Macedon's.

6.3 Deployability

Table 6.3 lists the requirements to start or join a session of a communication type in MayaJala. MayaJala does not require any special support from the network or the operating system. For machines behind a firewall it requires that the MayaJala port to be opened. The biggest deployability problem is on machines behind NAT boxes. Note that in the current prototype the session leader also acts as the default resolver for the communication type. Therefore, non session-leaders must contact the session leader to obtain the agent clone. The session leader must be on a machine that can be readily contacted from any other ma-

	Required support
Network	No special support needed.
Operating System	No special support needed.
Firewall	Ability to open connections to the MayaJala port. The default port in the current prototype is 2003.
NAT	In the current prototype the session leader must be on a real IP address. Whether session followers can be behind a NAT box depends on the agent implementation.
Software	Requires Java runtime environment, and the MayaJala middleware. The current prototype also requires the agent class files in the *classpath*. However, the design allows for dynamic loading of classes across the network.
Third party	None needed. Only the processes participating in a session cooperate to implement the communication type.
Coordination required to select an implementation	None needed. The session leader takes the decision without any coordination.

Table 6.3: The requirements to deploy.

chine. Some agents also may want to contact each other from both directions. In such a case, NAT is a barrier to deploy these implementations.

Only the processes that are on the session participate in message forwarding and implementing the communication type. We do not require other third party machines or processes to forward messages on behalf of a session. This is different from multicast groups implemented over DHT's such as Scribe where nodes that are not part of the multicast session participate in maintaining the membership and message forwarding.

One of the main strengths of MayaJala with respect to deployability is that there is no coordination required to select an implementation for a session. The session leader selects the implementation as we have seen in the previous examples. No coordination is required to select an implementation. The clients are not even aware of the selected implementation. Also it is the session leader that takes the decision to change the communication type with a conforming type. All the required coordination to use an implementation is embedded in the port predicate.

These minimum requirements for deploying shows that MayaJala is highly deploy-

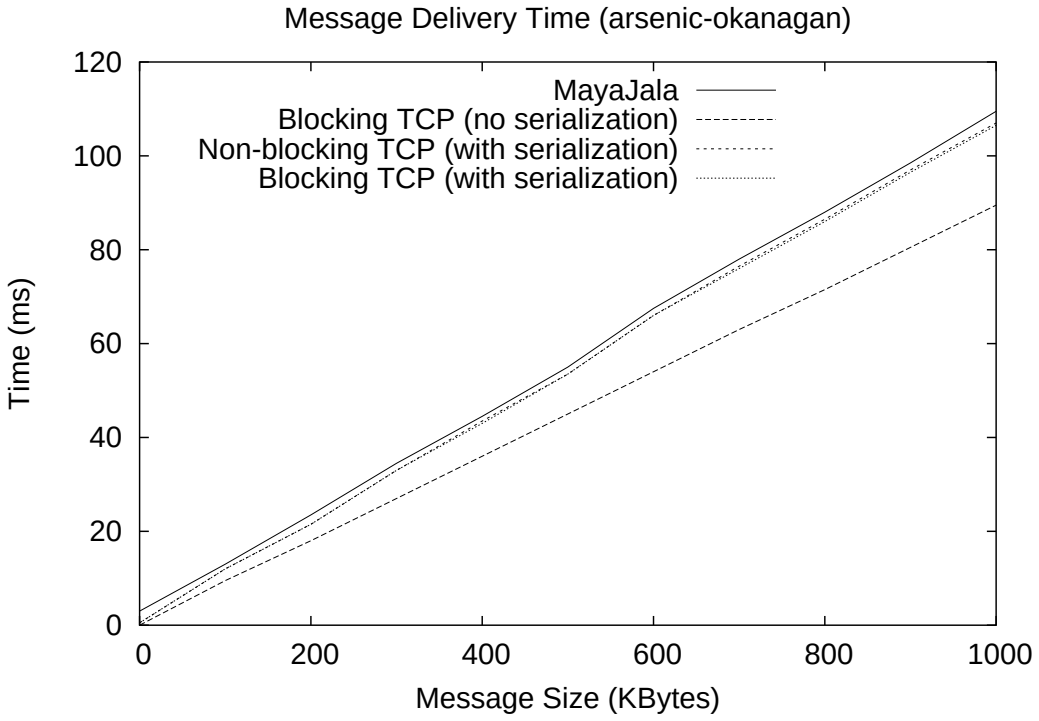

Figure 6.3: Message delivery time.

able. The major problem in the current prototype with respect to deployability is NATed environments. We designed MayaJala to be deployed over the Internet and machines behind NAT boxes are not exactly on the Internet. However, for complete deployability to be achieved this problem should be addressed. Note that the MayaJala design allows different components to be on different machines. One solution to the NAT problem is to place the router outside the firewall and the other components inside.

6.4 Overhead interposed by MayaJala

To evaluate the overhead added by MayaJala we measured message delivery time between two machines, arsenic and okanagan. arsenic runs Linux kernel version 2.6 and okanagan runs kernel version 2.6. arsenic is a dual processor (2.40 GHz Intel Xeon) machine with 512KB of cache and 1GB of memory. okanagan is a quad processor (2.66 GHz Intel Xeon)

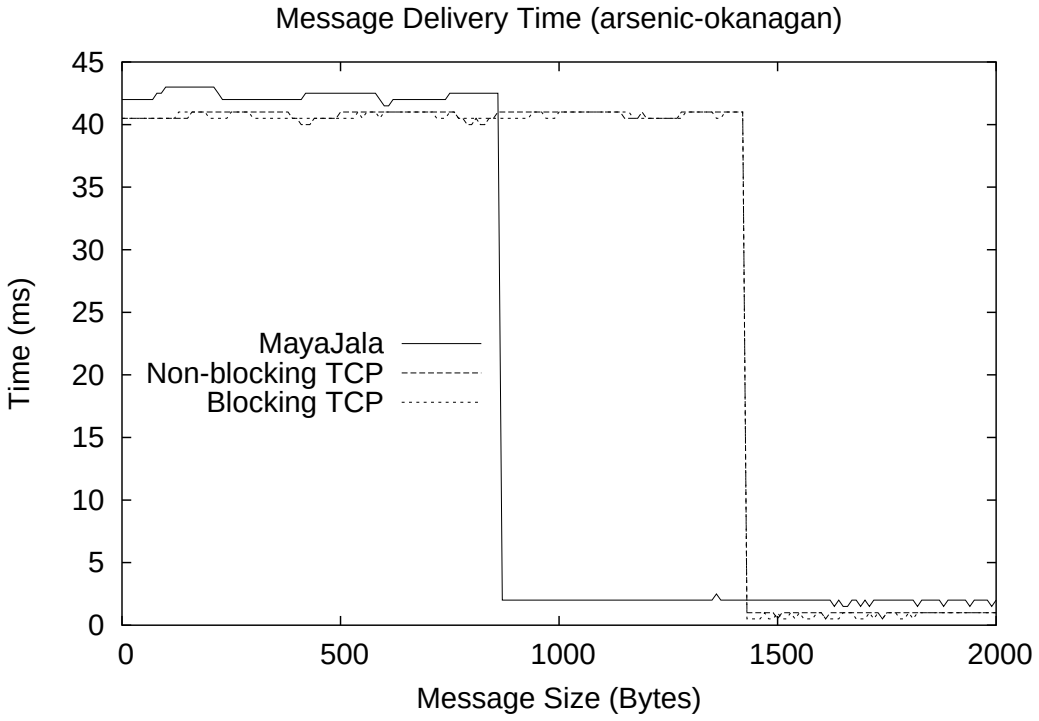

Figure 6.4: Message delivery time (small messages).

machine with 512KB of cache and 4GB of memory. The two machines are two hops away in two 100Mbps LANs separated by a latency of 0.6ms. The message delivery times were measured by echoing back messages sent by okanagan from arsenic. The results are given in the graph in Figure 6.3.

We compare message delivery time between two applications over MayaJala and also over TCP in three different modes, blocking with serialization, non-blocking with serialization, and blocking without serialization. The cases that use serialization send and receive objects and the case that does not use serialization sends directly from a ByteBuffer over a Java SocketChannel. Note that the object of size zero (a byte array) results in a non-zero length serialized message, while a zero length ByteBuffer does not result in message transfer over the network. The non-serialized case represents the best possible message delivery time over TCP.

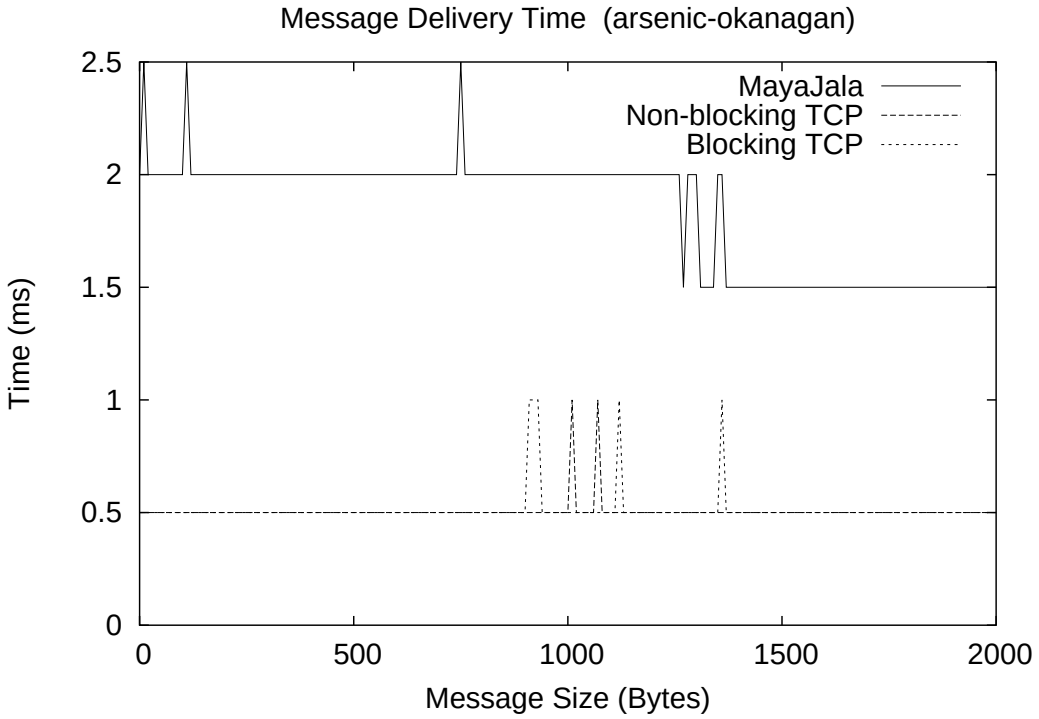

Figure 6.5: Message delivery time (with Nagle disabled).

Figure 6.4 shows the results of the same experiment for small messages in the range of 0 to 2 KBytes. Note that all three cases use serialization. The drop in message delivery time around 900 bytes for MayaJala and around 1400 bytes for other two cases is due to the Nagle algorithm. Figure 6.5 shows the message delivery time with the Nagle algorithm disabled. A MayaJala message carries a header for routing purposes. The information in the header can be represented using less than 32 bytes, however the Java object that represents the header serializes into 549 bytes. The effect of this 549 bytes can be clearly seen in Figure 6.4. When the Nagle algorithm is in effect this overhead works for MayaJala rather than against it. Figure 6.4 highlights the fact that there are artifacts, such as the Nagle algorithm, that we take for granted and have far greater impact on performance than the overhead of MayaJala.

We also tested MayaJala over two nodes on the Emulab testbed and the results are

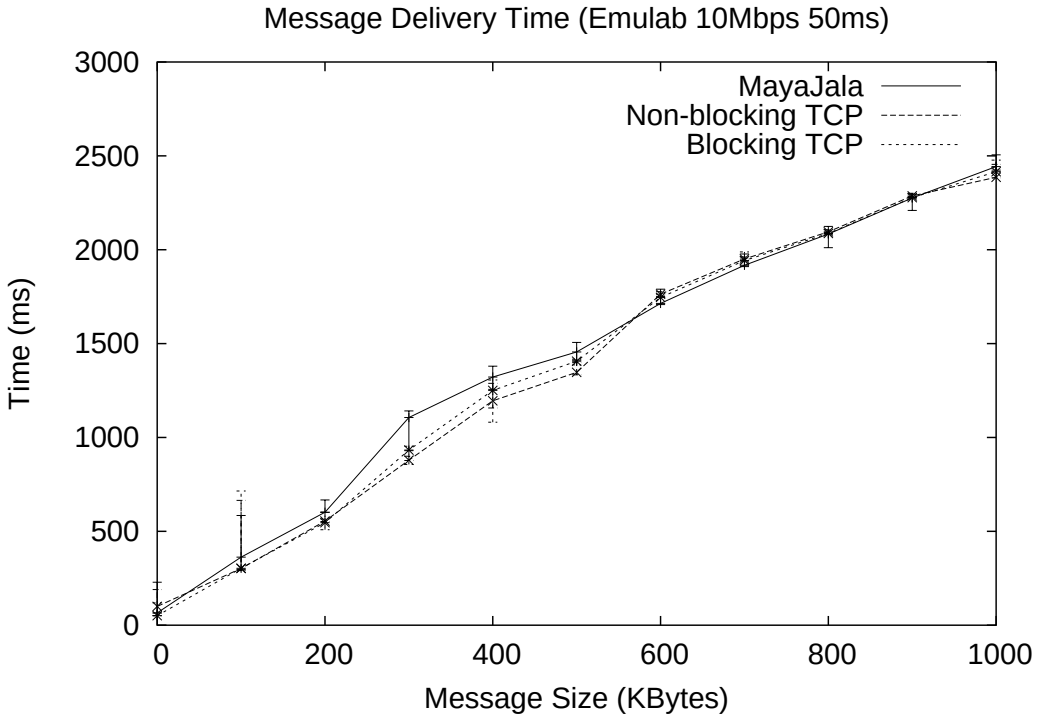

Figure 6.6: Message delivery time (Emulab).

shown in Figure 6.6. The two Emulab machines that we used for this experiment were running Linux kernel version 2.4.20. Each machine has a Pentium III processor (600 MHz) and 256 MB of memory. Object serialization time on the JVM on these machines are roughly 4 times slower than that on arsenic. This affects the serialization time of the header and hence the overhead. Even under these adverse circumstances, the message delivery time on MayaJala is within the margin of error of the baseline cases.

Each agent keeps a shadow routing table and applies modifications to this local table before updating the main routing table after a batch of modifications. Figure 6.7 shows the growth of the serialized size of a routing table with the number of routing rules in it. Each routing rule contains one output link. A routing table without any rules serializes into 147 bytes. A routing table that contains one routing rule serializes into 538 bytes and each additional routing rule adds 83 bytes. It takes less than 1 ms to send an update of a routing

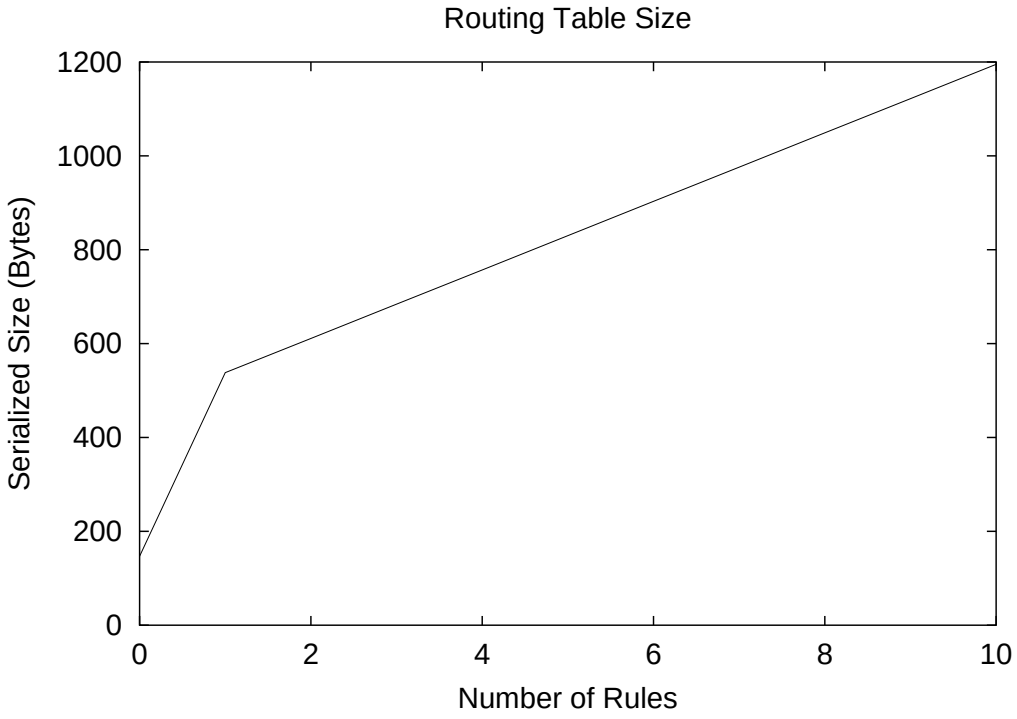

Figure 6.7: Serialized size of the routing table.

table that contains 10 routing rules to the router.

6.5 Summary

We showed that the use of communication types and MayaJala results in simple application code. MayaJala allows implementations and communication types to be changed with minimal code change at the session leader and without any code change at the non session-leaders. MayaJala also supports simple implementation of agents. MayaJala and agents are deployable and no coordination is required to select an implementation of a communication type or to change the communication type. When building MayaJala we chose fast prototyping over optimization. We make full use of the Java language features to do so. The above results show that even without striving for optimization we have built a prototype with minimal overhead.

Chapter 7

Conclusions and Future Work

In Chapter 1 we put forward the following thesis.

A broad class of multiparty-communication paradigms can be implemented by a common API and can be described by a formal model. Doing so simplifies an application's choice of communication library by syntactically de-coupling applications from communication libraries and by clarifying the semantic interdependence between the applications and libraries.

In this chapter we revisit the thesis in the light of the work done and the results obtained to make conclusions. Then the contributions from this work are presented. Finally, we present future work that are beyond the scope of the thesis but that will enhance this work and make the artifact, MayaJala, more useful.

7.1 Conclusions

This thesis has documented that there is a large spectrum of multiparty communication paradigms and a vast design space in implementing each of these communication paradigms over the global Internet. We assumed that these communication paradigms are important for distributed applications. We noted that lack of a naming system, a common API, and a deploying system hinders the exploration of the design space independent of the applications that want to use these communication paradigms. We presented MayaJala as a solution to

these issues.

MayaJala is based on the novel idea of multiparty communication types. We used an abstract model as a vehicle to present this idea. Through the model we showed that the multiparty communication type system provides a precise naming system for multiparty communication paradigms. The communication type system also gave rise to a common API that is suitable for all the multiparty communication types that can be defined using the model. This common API is a very powerful tool in the sense that we do not have to standardize an API for each and every communication type separately. This allows inventors to come up with new communication types and multiple implementors to implement them without any coordination and it also allows applications to use them without any coordination with the implementors. These features also indicate that the implementations of communication types can be very modular. According to Baldwin and Clark "A complex engineering system is modular-in-design if (and only if) the *process of designing it can be split up and distributed across separate modules,* that are coordinated by design rules, not by ongoing consultation among designers" [5]. Liskov and Guttag also give a similar definition [62]. The idea of communication types allows complex communication to be captured in a modular fashion with the name of the communication type itself providing the specification for implementations.

The notions of communication type equivalence and communication type conformance enhance the usefulness of the idea of communication types. The idea of communication type equivalence gives the flexibility to applications programmers to express the communication pattern required by the application in a most natural way to the application; this can be matched to an implementation using communication type equivalence. We showed that the notion of communication type conformance allows an implementation of a communication type to be replaced with not just another implementation of the same communication type, but also with other conforming communication types.

The fact that communication types do not have a programming language representation seems a hindrance at first glance. However, the idea of multiparty communication

types extends beyond the program text and captures global properties that are required for correct operation; programming language representations cannot capture these properties. The communication type system is another tool, such as design patterns, at the disposal of distributed-application developers.

The MayaJala middleware provides a system that facilitates the implementation and deployment of multiparty communication types. We showed that communication types implemented using MayaJala can be deployed without any coordination of the processes participating in a session. We also demonstrated that an implementation of a communication type can be changed with just a single line of code in the code of the session leader and without any change in the code of the non session-leaders.

On MayaJala communication types are implemented using overlay networks. We showed that MayaJala allows concise implementations of communication types (agents). The agent implementations on MayaJala are comparable in conciseness to implementations that use a domain specific language for building overlays. The overhead interposed by MayaJala is very minimal despite the fact that the prototype that was used for evaluation was implemented in Java; we make full use of the Java language features for fast prototyping without aiming for optimization. We showed that there are artifacts, such as the Nagle algorithm, that we take for granted, but have far greater impact on performance than MayaJala.

The above mentioned observations lead us to the conclusion that the thesis has been demonstrated. We demonstrated that applications can switch implementations of communication types without any code change in the applications. This shows that applications are free of syntactic dependence on implementations. This is possible because of the common API. We demonstrated that an implementation of a communication type can be switched with an implementation of a different, but conforming, communication type. This is possible because communication types capture the semantic interdependence between applications and implementations of communication types. Also the modular implementations of communication types demonstrate that the semantic interdependence is sufficiently cap-

tured by communication types to allow for independent implementations of communication types and applications. We also demonstrated that the ideas developed through the communication type system can be used in practice with minimal cost to the applications.

7.2 Contributions

There are four contributions from this work.

1. **The idea of multiparty communication types:** The main pillar that this work stands on is the novel idea of multiparty communication types. This idea led to two other important concepts; communication type equivalence and communication type conformance. The MayaJala framework, which we used to demonstrate the thesis, is based on the idea of multiparty communication types.

2. **The design of MayaJala:** The design of MayaJala makes the idea of multiparty communication types concrete and accessible to distributed applications. MayaJala is designed in such a way that an application can use multiple communication types and multiple implementations of communication types. It clearly identifies the roles of application programmers and type programmers. The design also facilitates the deployment of implementations in application with zero coordination. While the design is described in terms of the prototype implemented in Java language it is not language specific.

3. **The abstract model:** We used an abstract model as a vehicle to present the idea of multiparty communication types. While it is the idea that is important the model itself is also useful and can be improved to accommodate more communication types.

4. **The prototype implementation of MayaJala:** The prototype implementation of MayaJala helped us to demonstrate the use of communication types and carry out evaluations, The prototype itself is useful and can be used to build applications. The

prototype also can act as a reference implementation for future implementations in other languages.

7.3 Future work

While we demonstrated the thesis successfully using the artifact and the model that we developed there are certain limitations in the artifact and the model. Further work is required to address these issues. We discuss future works under three headings; improvements to the communication type system, improvements to the MayaJala middleware, and further evaluation.

7.3.1 Improvements to the multiparty communication type system

In Chapter 4 we used an abstract model to introduce the idea of multiparty communication types. While it served its purpose this model is not complete by any means.

As we noted in Chapter 4 there are desirable communication paradigms that the abstract model cannot capture. For example, the model cannot describe a communication type that imposes ordering between the messages sent and received by a given process. The model also cannot describe communication types that "eventually" deliver messages. We expect that this can be solved by extending the model to use temporal logic. Another drawback of the model is that it allows definition of "bogus" communication types; these are the communication types with message predicates that cannot be guaranteed in a meaningful manner under its port predicate. Further research is required to understand these issues better and come up with solutions for them.

It would be convenient if it is possible to check for conformance and equivalence of any given two communication types mechanically. However, we expect that this task to be an impossible one given the fact that predicate logic does not constitute a decidable system. However, perhaps a heuristics based approach can be used to help the processes of finding equivalent or conforming communication types.

7.3.2 Improvements to the design and implementation of MayaJala

There are several features in the design of MayaJala that have not been completely implemented in the prototype. One such feature is the link monitor. In the current version of the prototype the link monitor is a place holder. As we mentioned in Chpater 5 measuring and monitoring Internet paths is a research area in its own right and is beyond the scope of this work. It is possible to use the algorithms and techniques developed through the current research in this area to implement the link monitor. One major problem is to select the most suitable technique. This requires comparison of the existing techniques and matching them to the requirements of MayaJala. Further research is required to understand and control the impact and overhead of the measurement techniques on the operation of agents and applications.

Another feature that is missing in the current prototype is the ability to load the agent classes dynamically. The current version locates the agent objects for the sessions and downloads them to the local MayaJala instance, but it does not locate and download the classes of these objects. This is a useful feature, but it also raises several security issues such as how to determine the trustworthiness of the class and how to sandbox agents.

Another limitation of MayaJala is that the communication types are not embedded in the agents. As we mentioned before it may not be possible to compare communication types mechanically in all cases, but it is still useful if the communication types can be embedded in the agents. A database of agents which stores them against their communication types and which allows users to query for agents that implement a given communication type is also a useful feature.

MayaJala design allows for different classes of routing rules. The current implementation has only two such classes; one that forwards messages to all the matched links and one that forwards messages to a randomly selected link out of the matched set. Further experience in implementing different types of agents is required to understand the set of useful routing rule classes.

There is a body of work that uses distributed hash table (DHT) based overlay net-

works to implement multicast and anycast. While it is possible to build DHTs in MayaJala, we have not explored this avenue. In the MayaJala model only the nodes that are in the communication session participate in the overlay while in DHT based multicast implementations the multicast trees are built on a bigger *universal* DHT. One of the criticisms leveled against DHTs is the lack of such a universal DHT [50]. If such a universal DHT is available an interesting research avenue would be to use it as a network that sits below the MayaJala router (as the IP network in the current version) that provides richer communication facilities.

The current implementation is tied to the Java language and uses Java specific features. Porting MayaJala to other languages is a future direction in its evolution even though this is not strictly a research issue. A more challenging approach is to allow MayaJala instances implemented in different languages to interoperate. For example, the current implementation expects to download agents from a name resolver to join a session and we expect the agents to be implemented in Java. To allow interoperability we must define a common format to describe the state to be installed in the new node.

7.3.3 Further evaluation

This work was based on the assumption that multiparty communication paradigms are useful for distributed-application programmers. We assume that such communication paradigms simplifies applications. While these assumptions are reasonable, it would be useful to carry out further research to ascertain the usefulness of the multiparty communication types (and hence the communication paradigms) to the applications. Such research requires availability of large number of communication types and user studies. MayaJala provides the basic infrastructure to carry out such research and we believe that MayaJala will spur implementations of a large number of communication types. To encourage implementations of communication types and use of them in applications we plan to release MayaJala as an open source project.

User studies can also shed light on how multiparty communication types affect the

programming practices of distributed-application programmers. Communication types have properties that extend beyond the program text. It would be interesting to study the effects of such aspects on the cognitive model of the programmers.

Similar experiments can be carried out to ascertain the usefulness of the services provided by the router to type programmers. We showed that agents can be designed concisely using MayaJala, but we did not show that any programmer can come up with similarly concise implementations. A user study on type programmers could shed light on this issue.

As type programers use MayaJala to explore the design space we expect that more interesting research questions will arise. We expect that our plan of releasing a production version of MayaJala as an open source project will encourage the exploration of design space resulting in more research problems.

The above discussion shows the breadth of the research area that can be based on the artifacts that we produced in this work. The MayaJala middleware can facilitate all these experiments.

Bibliography

[1] R. Ben Abbou, Amine Benkiran, and Jean-Pierre Courtiat. Formal validation of a multicast transport protocol. In *Proceedings of the International Symposium on Computers and Communications*, pages 642–647. IEEE, 2001.

[2] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 327–336. IEEE Computer Society, 2000.

[3] Mostafa H. Ammar. Probabilistic multicast: Generalizing the multicast paradigm to improve scalability. In *Proceedings of the 13th Annual Joint Conference of the IEEE Computer and Communications Societies on Networking for Global Communciation. Volume 2*, pages 848–855, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[4] B. R. Badrinath and Pradeep Sudame. Gathercast: The design and implementation of a programmable aggregation mechanism for the internet. In *Proc. IEEE International Conference on Computer Communications and Networks (ICCCN)*, pages 206–213, October 2000.

[5] Carliss Y. Baldwin and Kim B. Clark. Modularity in the design of complex engineering systems. Working Paper Series 04-055, 2004, Harvard Business School, January 2004.

[6] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core based trees (CBT). In *Proceedings of the SIGCOMM '93*, pages 85–95. ACM, 1993.

[7] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217. ACM Press, 2002.

[8] Bryan Bayerdorffer. Distributed programming with associative broadcast. In *Proceedings of the 27th Annual Hawaii International Conference on Systems Sciences volume 2*, pages 353–362. IEEE, 1994.

[9] Bryan Carl Bayerdorffer. *Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems*. PhD thesis, The University of Texas at Austin, 1993.

[10] Gregory D. Benson, Cho-Wai Chu, Qing Huang, and Sadik G. Caglar. A comparison of MPICH allgather algorithms on switched networks. In *PVM/MPI*, pages 335–343, 2003.

[11] Merrie Bergman, James Moor, and Jack Nelson. *The Logic Book: Second Edition*. McGraw-Hill, 1990.

[12] S Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Fei Zongming. Application-layer anycasting. In *Proceedings of the Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM 97*, pages 1388 –1396. IEEE, 1997.

[13] Supratik Bhattacharyya. An overview of source-specific multicast (SSM). IETF request for comments (RFC) 3569, July 2003.

[14] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–160, 2000.

[15] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[16] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[17] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distributed and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.

[18] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[19] Kenneth L. Calvert, James Griffioen, Amit Sehgal, and Su Wen. Concast: Design and implementation of a new network service. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, pages 335–344. IEEE Computer Society, 1999.

[20] Casey Carter, Seung Yi, and Robin Kravets. ARP considered harmful: Manycast transactions in ad hoc networks. In *WCNC 2003 - IEEE Wireless Communications and Networking Conference*, pages 1801–1806. IEEE, 2003.

[21] Casey Carter, Seung Yi, Prashant Ratanchandani, and Robin Kravets. Manycast: Exploring the space between anycast and multicast in ad hoc networks. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 273–285. ACM Press, 2003.

[22] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. In *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, October 2002.

[23] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scalable application-level anycast for highly dynamic groups. In *Proceedings of the Fifth International Workshop on Networked Group Communications (NGC'03)*, September 2003.

[24] Youngsu Chae, Ellen W. Zegura, and Haris Delalic. PAMcast: Programmable anymulticast for scalable message delivery. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 25–36, June 2002.

[25] Shun Yan Cheung and Akhil Kumar. Efficient quorumcast routing algorithms. In *Proceedings of the 13th Annual Joint Conference of the IEEE Computer and Communications Societies on Networking for Global Communciation. Volume 2*, pages 840–847, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[26] Clayton M. Christensen. *Innovator's Dilemma*. HarperBusiness, 1997.

[27] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proceedings of the ACM SIGMETRICS*. ACM, June 2000.

[28] National Research Council. *Looking Over the Fence at Networks: A Neighbor's View of Networking Research*. National Academy Press, Washington, D.C, 2001.

[29] S. J. Creese and Joy N. Reed. Verifying end-to-end protocols using induction with CSP/FDR. In *IPPS/SPDP Workshops*, pages 1243–1257, 1999.

[30] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[31] S. Deering and R. Hinden. RFC 2460: Internet protocol, version 6 (IPv6) specification, December 1998.

[32] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989.

[33] Stephen E. Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. The PIM architecture for wide-area multicast routing. *IEEE/ACM Trans. Netw.*, 4(2):153–162, 1996.

[34] Michael Evangelist, Nissim Franzes, and Shmuel Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(2):1417–1426, 1989.

[35] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science*, 1908:346–, 2000.

[36] Amy P. Felty, Douglas J. Howe, and Frank A. Stomp. Protocol verification in nuprl. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 428–439, London, UK, 1998. Springer-Verlag.

[37] Nissim Francez and Brent Hailpern. Script: A communication abstraction mechanism. In *PODC*, pages 213–227, 1983.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[39] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[40] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[41] Tian He, John A. Stankovic, Chenyang Lu, and Tarek Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 46, Washington, DC, USA, 2003. IEEE Computer Society.

[42] Mohamed Hefeeda, Ahsan Habib, Boyan Botev, Dongyan Xu, and Bharat Bhargava. Promise: peer-to-peer media streaming using collectcast. In *Proceedings of the eleventh ACM international conference on Multimedia*, pages 45–54. ACM Press, 2003.

[43] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[44] Hugh W. Holbrook and David R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. *SIGCOMM Comput. Commun. Rev.*, 29(4):65–78, 1999.

[45] Qingfeng Huang, Chenyang Lu, and Gruia-Catalin Roman. Spatiotemporal multicast in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 205–217, New York, NY, USA, 2003. ACM Press.

[46] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

[47] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 197–212. USENIX, 2000.

[48] D. Johnson and S. Deering. RFC 2526: Reserved IPv6 subnet anycast addresses, March 1999.

[49] Yuh-Jzer Joung and Scott A. Smolka. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems.*, 16(3):954–985, 1994.

[50] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring adoption of DHTs with openhash, a public DHT service. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, February 2004.

[51] Dina Katabi and John Wroclawski. A framework for scalable global IP-Anycast (GIA). In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–15. ACM Press, 2000.

[52] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press, 1999.

[53] Charles Krasic, Kang Li, and Jonathan Walpole. The case for streaming multimedia with TCP. Technical Report CSE-01-003, OGI CSE Technical Report, March 2001.

[54] Thomas Kunz and Michiel F. H. Seuren. Fast detection of communication patterns in distributed executions. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 12. IBM Press, November 1997.

[55] G. Kwon and J. Byers. ROMA: Reliable overlay multicast with loosely coupled TCP connections. In *Proc. of IEEE INFOCOM*, Hong Kong, March 2004.

[56] Kevin Lai and Mary Baker. Measuring bandwidth. In *INFOCOM*, pages 235–245, 1999.

[57] Karthik Lakshminarayanan, Ananth Rao, Ion Stoica, and Scott Shenker. Flexible and robust large scale multicast using i^3. Technical Report UCB/CSD-02-1187, Computer Science Division (EECS) University of California Berkeley, Berkerly, June 2002.

[58] LAM Team. LAM/MPI. http://www.lam-mpi.org/.

[59] Jong-Kun Lee and Kwang-Hui Lee. Modeling of the multicast transport protocols using petri nets. In *Proceedings of IEEE Singapore International Conference on Networks and International Conference on Information Engineering '95*, pages 106–110, 1995.

[60] Jörg Liebeherr, Jianping Wang, and Guimin Zhang. Programming overlay networks with overlay sockets. In *Networked Group Communication*, pages 242–253, 2003.

[61] H. Paul Lin. Modeling a transport layer protocol using first-order logic. In *SIGCOMM*, pages 92–100, 1986.

[62] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[63] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, pages 37–42. IEEE CS Press, 2001.

[64] Carolos Livadas and Nancy A. Lynch. A formal venture into reliable multicast territory. In *FORTE*, pages 146–161, 2002.

[65] Message Passing Interface Forum. MPI: A message-passing interface standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, June 1995.

[66] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, July 1997.

[67] Chris Metz. IP Anycast: point-to-(any) point communication. *IEEE Internet Computing*, 6(2):94 –98, March 2002.

[68] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 217–231. ACM Press, 1999.

[69] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 153–167. ACM Press, 1996.

[70] Akihiro Nakao, Larry Peterson, and Andy Bavier. A routing underlay for overlay networks. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 11–18. ACM Press, 2003.

[71] Julio C. Navas and Tomasz Imielinski. GeoCast - geographic addressing and routing. In *MobiCom '97: Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking*, pages 66–76, New York, NY, USA, 1997. ACM Press.

[72] Open DHT Team. Open DHT. http://openhash.org/.

[73] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. Internet Engineering Task Force: RFC 1546, November 1993.

[74] Jan Pedersen and Alan Wagner. Correcting errors in message passing systems. In *Proc. of High-Level Parallel Programming Models and Supportive Environments : 6th International Workshop, HIPS 2001*. Springer Verlag, April 2001.

[75] Dimitris Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, San Francisco, CA, USA, March 2001.

[76] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[77] PlanetLab Consortium. PlanetLab. http://www.planet-lab.org/.

[78] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software-Practice and Experience.*, 21(1):91–118, 1991.

[79] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Computer Communication Review.*, 31(4):161–172, 2001.

[80] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 14–29. Springer-Verlag, 2001.

[81] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Technical Conference, General Track*, pages 127–140, 2004.

[82] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI*, pages 267–280, 2004.

[83] Gruia-Catalin Roman, Octav Chipara, Chien-Liang Fok, Qingfeng Huang, and Chenyang Lu. A Unified Specification Framework for Spatiotemporal Communication. Technical Report WUCSE-03-66, Washington University, Department of Computer Science and Engineering, St. Louis, Missouri, 2003.

[84] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[85] Jeffrey M. Squyres. *A Component Architecture for the Message Passing Interface (MPI): The System Services Interface (SSI) of LAM/MPI*. PhD thesis, University of Notre Dame, 2004.

[86] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, third edition, 2003.

[87] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, August 2002.

[88] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[89] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference '03*, Miami, Florida, October 2003.

[90] Rajeev Thakur and William Gropp. Improving the performance of collective operations in MPICH. In *PVM/MPI*, pages 257–267, 2003.

[91] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 80–89. ACM Press, 1995.

[92] D. Waitzman, C. Partridge, and S. Deering. RFC 1075: Distance vector multicast routing protocol, November 1988.

[93] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243. ACM Press, 2001.

[94] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A design framework for highly concurrent systems. Technical report, University of California at Berkeley, 2000.

[95] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *Proceedings of the International Conference on Open Architectures and Network Programming (OPENARCH)*, 1998.

[96] Jaehee Yoon, Azer Bestavros, and Ibrahim Matta. Somecast: A paradigm for real-time adaptive reliable multicast. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 101. IEEE Computer Society, 2000.

[97] Qianfeng Zhang, Chamath Keppitiyagama, and Alan S. Wagner. Supporting MPI collective communication on network processors. In *CLUSTER*, pages 75–82, 2002.

[98] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, University of California at Berkeley, 2001.

[99] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *11th International workshop on on Network and Operating Systems support for digital audio and video*, pages 11–20. ACM Press, 2001.