MayaJala: A Framework for Multiparty Communication

Chamath Keppitiyagama Norman C. Hutchinson

Department of Computer Science
University of British Columbia
Vancouver, B.C, Canada, V6T 1Z4
E-mail: {chamath,norm}@cs.ubc.ca

Abstract

The availability of higher bandwidth at the end-points has resulted in the proliferation of distributed applications with diverse communication needs. Programmers have to express the communication needs of such applications in terms of a set of point-to-point communication primitives. Such an approach has several disadvantages. These include the increase in development time, the difficulty in getting the complex communication structure right and the redundancy in several developers doing the same work to achieve the same communication pattern. If common communication patterns are available as well defined communication types these problems can be avoided. A similar approach has already been used in the parallel programming domain; message passing parallel programs use collective communication primitives to express communication patterns instead of composing them with point-to-point communication primitives. This is not the case for distributed programs in general. In this paper we discuss different multiparty communication types and also a framework to implement and make them available to distributed programs.

1. Introduction

Components of distributed applications need to communicate with each other and often these interactions exhibit interesting patterns. In this paper we argue for *communication types* for these *communication patterns*. Before proceeding any further, we must clarify these terms. We use the term *communication pattern* to denote a description of the communication and the term *communication type* to denote a higher level abstraction that names the communication pattern. For example, *multicast* is a communication type that represents the communication pattern of one node sending the same message to all the other nodes. The communication types are independent of any particular implementation or a particular interface. We also use the term *communication primitives* to denote well defined communication interfaces such as MPI [17].

There are several advantages of using communication types and implementations of communication types in different phases of the development and the deployment of distributed applications, instead of describing the communication in terms of a set of point-to-point communication primitives.

- The design phase: Designers of the distributed application can express the communication patterns in the application in terms of higher level communication types. At this stage it is important to describe the communication at such a level of abstraction without committing to any implementation or any interface. Named communication types also facilitate the discussions among the members of the development team. For example, we use the communication type multicast to describe communication without describing the implementation of multicast—in other words we assume that multicast is available as a communication type.

- The implementation phase: At the implementation phase, the application programmers must commit to a particular interface for a communication type. There is a direct relationship between the communication type and the interface. However, they do not have to deal with the question of how to achieve (or implement) the communication pattern. They simply use a communication primitive that matches the required communication type to express the communication in the program. For example, MPI programmers can use `MPI_Bcast()` to describe the communication pattern of one node sending a message to all the other nodes without any reference to how this is done.

- The deployment phase: The application programmers do not have to commit to any particular implementation of the communication type. This improves the portability of the application. The same communication type can be implemented differently in a different environment transparently to the programmer. The program can be linked to a different implementation to take advantage of the features of that implementation. MPI is a fine example. Parallel programmers write message passing parallel applications using MPI communication primitives and these applications can be linked with different implementations of MPI in different environments. In particular, implementations of MPI collective communication routines take advantage of the communication facilities of different environments. This happens transparently to the application. On the other hand, once a communication type has been implemented it can be used in different applications. This promotes reusability.

- The debugging phase: Getting the communication right in parallel and distributed applications is not an easy task. Kunz et al. [16] describe the use of *communication patterns* for visualizing and debugging distributed applications. In their approach programmers give the visualization tool a description of the pattern and a trace of the application execution is examined to find the pattern. Pedersen and Wagner [20] also describe a debugging method to infer the plausible communication patterns in parallel programs. Both these approaches use the communication patterns in the debugging stage. A better approach would be for the programmer to use the required communication pattern, by using a communication type, in the program itself.

Apart from multicast there are no other multiparty communication types in general use in wide area distributed applications. Even the familiar communication type multicast has not been used widely because of the lack of support from IP networks. The IP protocol supports multicast [11], but it is not widely available because of deployment problems. However, we observe recent efforts in providing multicast support for distributed applications at the application level. The MBone overlay network also had some success in deploying multicast, but access to MBone is not yet universal.

There are several reasons for the lack of demand for multiparty communication types in distributed applications. The majority of the traditional Internet applications can be categorized as client/server applications. These applications have simple point-to-point communication patterns. There is not a big demand for multiparty communication types from these applications. Even some Internet applications that have natural multicast communication patterns have resigned to use multiple point-to-point communication rather than multicast. A case in point is some live Internet radio services. This made sense in a situation where the servers had higher bandwidth links to the Internet while the clients were mainly dial-up clients without multicast support at the IP level. Low bandwidth connections to the homes did not provide an environment for communication intensive applications like video conferencing. When only a handful of applications are using a given communication pattern there is little payoff in investing the effort to implement the pattern as a communication type.

Another reason for not having MPI-like communication primitives for distributed applications is the difficulty in defining the semantics. Take for example barrier communication. What is the semantics of a barrier when some nodes could fail and still others could progress? What is the semantics of reduction under the same conditions? Does it make any sense at all to use these communication types in such a situation? Also the semantics of MPI-like communication types do not fit the bill of communication in distributed applications in general—we discuss this in detail in the following section.

The availability of higher bandwidth and processing power at the traditional client ends have resulted in demand for new types of applications. Fomenkov et al. [13], in their study on the Internet traffic from 1998 to 2001, note that the share of world wide web (WWW) traffic, which was the dominant traffic type, reached a peak during late 1999 and early 2000.[1] They note that this coincides with the appearance of new peer-to-peer applications. Even the WWW traffic contains traffic from peer-to-peer applications. In fact, a study [22] on incoming and outgoing Internet traffic at the University of Washington concludes that peer-to-peer traffic accounts for the majority of the HTTP bytes transfered. Also, applications like video conferencing and e-learning, which need richer communication patterns, are becoming more and more popular.

Despite those problems mentioned before, the need for different multiparty communication types has not gone entirely unnoticed. Several researchers have argued for and suggested different communication types for distributed applications. Anycast [19], Manycast [6], Concast [5], PAMcast [7], Gathercast [2], Quorumcast [8] and SomeCast [24] are some of them. Most of these communication types are certainly not new. The novelty is the explicit naming of the communication pattern and the use of them as a communication type (or providing them as services).

There are problems associated with this approach of providing different multiparty communication types in isolation. First, users need completely different systems for different communication types. This increases the complexity of the application and the task of the application programmer. Second, we observe that there is functionality common

[1] The university of Waterloo traffic reports, available at http://ist.uwaterloo.ca/cn/Stats/ext-prot.html, also shows that the traffic other than the web traffic taking the dominant share of the bandwidth since 2002.

to all these communication types. The above approach fails to recognize this and significant effort and resources are invested in duplicating the common functionality. Third, there is no clear plan for future communication types. It is better to study multiparty communication types in general and provide a framework to support all existing and anticipated future communication types. We present such a framework called MayaJala.[2] Our framework consists of following components.

- **A type system:** The type system provides an implementation and interface agnostic method to specify communication types. This is useful at the designing stage of the distributed applications, when the designers are not committed to any interface or an implementation, to precisely specify the communication of the application.
- **A programming model:** The programming model describes how the types are implemented (*type programming*) and how the types are used in the application (*application programming*).
- **A middleware system:** The middleware system provides infrastructure to support type programming, application programming, and the deployment of implementations of communication types.

The rest of the paper is organized as follows. In Section 2 we discuss related work. We present the communication type system in Section 3 and in Section 4 we discuss the programming model. In Section 5 we present the middleware system. Finally, we give conclusions in Section 6.

2. Background and related work

The motivation for different communication types comes from the collective communication primitives in the Message Passing Interface (MPI) [17, 18]. MPI provides a large collection of collective communication primitives. As the name suggests, the MPI collective communication routines are truly collective—all the nodes in the group (communicator [17]) must call the routines in unison. This cohesive communication model is well suited to parallel applications. After all, MPI is 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 [18] 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. Again, as an example, compare this with the group management of IP multicast. 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 [12]. This defeats the goal of many distributed applications, which are distributed to avoid a single point of failure. Although MPI provides motivation for this work, we recognize that the communication requirements of distributed applications are different from those of parallel applications.

In the distributed computing world multicast is a well known communication type. Programmers have used multicast as a named type for a long time. However, using multicast over the Internet is a challenging task. Even though multicast is a part of the IP protocol suite [11], it is not universally available due to deployability and scalability problems. One solution to this problem is the MBone overlay network. MBone connects IP multicast capable “islands” using an overlay network. Another solution is to use application-level multicast. Several projects have implemented application-level multicast systems. These include Narada [9], Overcast [14], ALMI [21] and Bayeux [25]. These projects are different in terms of tree construction methods, optimization goals, target applications and the architecture.

Anycast, even though not as popular as multicast, is another well known communication type in distributed applications. In Anycast, a message is sent to *any* one member of a group—usually the nearest according to some metric. It is a useful communication type for server selection. For example, if the same service is available from different servers (say, mirror sites) a client could anycast a message to the group formed by those servers. Anycast is also included in the IPv6 protocol suite [10, 15]. While Anycast has often been used to describe the communication pattern, it is rarely used as a communication type in applications. This is mainly due to the lack of availability of Anycast as a service. Castro et al. [6] and Bhattacharjee et al. [3] describe application-level Anycast implementations as a solution to this problem.

Many-to-one communication, which is the inverse of multicast, is also an important communication pattern. Concast [5] and Gathercast [2] are two implementations of this pattern as a network service in wide area networks. Calvert et al. [5] suggest the use of Concast for acknowledgment aggregation and other summarized feedback schemes. Badrinath et al. [2] identify sensor networks, situation awareness applications and responsive environments as some of the applications that could benefit by the Gath-

[2]The Sanskrit word Maya means illusion and the word Jala means networks—illusionary networks, or virtual networks. This name was selected because the middleware component of the framework is based on application level virtual networks.

ercast service. Even without the summarizing capabilities, these two schemes provide scalable systems because the gathering point does not have to be aware of the identity of the group members. This is analogous to the IP multicast situation in which the sender does not have to be aware of the group members.

Chae et al. [7] introduce Programmable Any-Multicast (PAMCast) as a generalization of Anycast and multicast. In PAMCast, a message is sent to m out of n group members. This pattern is similar to the Manycast communication type mentioned by Castro et al [6]. Chae et al. [7] identify parallel cache queries, parallel downloading and fault tolerant repositories as some of the applications that can benefit from this type of communication pattern. Cheung et al. [8] introduced a similar communication pattern, Quorumcast, almost a decade earlier. Quorumcast was introduced as a communication paradigm akin to quorum consensus synchronization. Probabilistic multicast [1] also has a similar communication pattern, but the set of nodes to receive the message is determined randomly.

Yoon et al. [24] introduce SomeCast as the communication paradigm of receiving from some members of a group of senders. They describe a reliable real time multicast scheme, where users could receive according to their Quality of Service (QOS) requirements using SomeCast.

3. Multiparty Communication Types

Loosely defined communication types could raise more questions than providing answers. Take for example the use of the communication type multicast. One question that is often raised with the use of that term is about the reliability of the communication. This question arises because we use the same term multicast, some times annotated with other terms, to describe more than one communication pattern. For example we use the same term to describe the communication pattern of one node sending the message and all the other nodes receiving it (reliable multicast) and the pattern of one node sending the message and some of the other nodes receiving it (best effort multicast). In fact, these are two different communication types. We can reserve the word multicast for the type that describes the former pattern and it seems *somecast* is appropriate for the latter. However, SomeCast [24] has already been used to describe a completely different communication pattern. This shows the folly of using everyday terms to describe communication types.

Similar problems are associated with the use of the term Anycast. In Anycast, is there a single recipient of the message or more than one recipient? Is the message delivery guaranteed? Different answers to these questions, in fact, reveal different communication types described under the common banner of Anycast.

The above discussion highlights two important concerns: we need a method to precisely define communication types and the use of everyday terms to describe communication types adds to the confusion. We address both these concerns by developing a communication type system based on familiar discrete mathematical objects. Another goal of the type system is to explore the spectrum of the possible communication types. The communication type system is an important component of the MayaJala framework.

3.1. The Type system

We define a communication type as follows. Let a message be an indivisible (within the communication system) data unit that a node (or a process) sends to another node. Denote a message using lower case letters (usually m) and refer to more than one message by using subscripts (e.g., $m_0, m_1 \dots m_n$). To identify different nodes, number them consecutively starting from zero. Associate a set of messages (possibly $\emptyset$) with each node i and denote this set by M_i. Message sets associated with all the nodes are given by a sequence, $M = \langle M_0, \dots, M_n \rangle$. Let the set of all such sequences be $\mathcal{S}$. A communication type $t(M, \overline{M})$ is a binary relation defined on $\mathcal{S}$, where $M, \overline{M} \in \mathcal{S}$. Informally, M is the sequence of message sets in the *system* before the communication and $\overline{M}$ is the sequence of message sets in the *system* after the communication.

The set of all communication types is $\mathcal{T}$. All communication types have the following property, which states that communication does not destroy messages.

$$\forall t \in \mathcal{T} \quad t(M, \overline{M}) \Rightarrow \forall i \; M_i \subseteq \overline{M_i}. \tag{1}$$

Some examples should explain the use of the type system better. Take multicast as the first example. We define the multicast type t_{mult} as follows:

$$t_{mult}(M, \overline{M}) \Leftrightarrow \exists k \; \forall i \; (\overline{M_i} = M_k \neq \emptyset) \wedge (i \neq k \Rightarrow M_i = \emptyset).$$

The interpretation of the above equation is as follows. If there is exactly one node with a non null message set in the before sequence (M) and exactly that message set appears in all the nodes in the after sequence ($\overline{M}$), then the predicate $t_{mult}(M, \overline{M})$ is true. There is no ambiguity and t_{mult} describes reliable multicast.

The best effort multicast type, t_{mult_best}, can be described as follows:

$$t_{mult\_best}(M, \overline{M}) \Leftrightarrow \exists k \; \forall i \; ((\overline{M_i} = M_k \neq \emptyset) \vee (\overline{M_i} = \emptyset)) \wedge (i \neq k \Rightarrow M_i = \emptyset).$$

t_{mult_best} describes the pattern where only one node has a non-null message set, say m, associated with it in the before sequence and if a node has non-null message set associated with it in the after sequence then it is equal to m.

The Anycast communication type, t_{any}, can be defined as follows:

$$t_{any}(M, \overline{M}) \Leftrightarrow$$
$$\exists k, j\ ((k \neq j \wedge (\overline{M_j} = M_k \neq \emptyset)) \wedge (\forall i\ i \neq k \Rightarrow M_i = \emptyset)).$$

This definition of Anycast allows more than one node to receive the message and insists that at least one node gets the message.

Note that these definitions of the communication types are completely independent of any notion of a particular implementation. For example, if multicast is defined as a node *sending* a message to all the other nodes, we are committed to an implementation. But, our type system defines a communication type as a predicate on the before and after conditions of the system, without any reference to how the messages are moved around.

One advantage of having this kind of formal type system is that we can identify uncommon communication types. For example we could have a flavor of all-to-all communication that exchanges messages with nodes that have a common message - *every node that has a common message exchanges the other messages*. Let's denote this type as t_x.

$$t_x(M, \overline{M}) \Leftrightarrow$$
$$\forall i, j\ M_i \cap M_j \neq \emptyset \Rightarrow M_i \subseteq \overline{M_j} \wedge M_j \subseteq \overline{M_i}.$$

Communication type t_x has application to distributed interactive applications like distributed games. Smed et al. [23] describe a technique called *interest management* used in such applications. In this technique, each node's interest on data is expressed as its *aura*. When *auras* of two nodes intersect they become aware of each other—in other words they exchange messages. The communication type t_x is a perfect fit for that. Assuming that the interest is represented by messages, a non null intersection of the message sets of two nodes indicates the intersection of *auras*. Then by the complete exchange of messages they become aware of each other.

One could argue that the above pattern could simply be achieved by nodes with common interest joining a common multicast group. But that is an implementation of the pattern and there can be many such implementations. In fact, Smed st al. [23] describe two possible implementations—a central *subscription manager* based implementation and a multicast based implementation. This again brings out an important aspect of the type system—it describes the communication pattern without any commitment to an implementation.

3.2. Type Equivalence

Infinitely many different logical formulas can be used to describe the same communication type. It is important that we are able to check whether two communication types are equal in all respects. We give the following definition to this end.

Two communication types are equivalent if their definitions are logically equivalent.

Let's take an example to explain this. Consider the type t_y defined as follows:

$$t_y(M, \overline{M}) \Leftrightarrow$$
$$\forall i, j\ M_i \nsubseteq \overline{M_j} \vee M_j \nsubseteq \overline{M_i} \Rightarrow M_i \cap M_j = \emptyset.$$

We can prove that the communication type t_y is equivalent to the communication type t_x, that we defined previously, by proving that their definitions are logically equivalent.

Proof. Let's start with the formula on the right hand side of the definition of t_y and apply logical equivalence laws to get a series of logically equivalent formulas.

$$\forall i, j\ M_i \nsubseteq \overline{M_j} \vee M_j \nsubseteq \overline{M_i} \Rightarrow M_i \cap M_j = \emptyset.$$

By applying the contrapositive law:

$$\forall i, j\ \neg(M_i \cap M_j = \emptyset) \Rightarrow \neg(M_i \nsubseteq \overline{M_j} \vee M_j \nsubseteq \overline{M_i}).$$

By applying DeMorgan's law:

$$\forall i, j\ \neg(M_i \cap M_j = \emptyset) \Rightarrow \neg(M_i \nsubseteq \overline{M_j}) \wedge \neg(M_j \nsubseteq \overline{M_i}).$$

This is equivalent to:

$$\forall i, j\ M_i \cap M_j \neq \emptyset \Rightarrow M_i \subseteq \overline{M_j} \wedge M_j \subseteq \overline{M_i}.$$

This is the formula on the right hand side of the definition of t_x. Therefore, $t_y \Leftrightarrow t_x$. □

This highlights another advantage of the type system. The communication pattern needed for an application can be described, using the type system, in the most natural way to the application. Different designers could define the same communication type differently. Once the required type is defined it could be compared against the already known and available communication types, using the logical equivalence rules, to find a matching communication type.

3.3. Multiparty Communication

The type system, as described above, is not restrictive enough to describe only multiparty communication patterns. It can also describe communication patterns that should be described as several independent patterns. Kunz et al. [16] also had the same problem and chose to restrict the communication patterns only to connected patterns. These are the communication patterns that connect all the nodes together (in their parlance "connectivity between all the strings"). In other words, connected patterns have connected communication graphs. However, such a definition could reject some patterns that we are interested in or recast them as something un-intended. Take for example Anycast. In Anycast, a node sends a message to any of the other nodes in the group. Ultimately this is just a point-to-point communication. Therefore, the communication pattern does not connect all the nodes. However, the recipient node is selected from the whole group and every node has the potential to be the recipient of the message. In other words the whole group participates in this pattern, even though only two nodes exchange messages. So we give the following restriction on the type system that filters out non-multiparty communication patterns.

If $t(M, \overline{M})$ is a multiparty communication type, then t must be defined over the whole set of nodes, using either universally or existentially quantified variables over the set of all the message sets (or the node identifiers). Identification of nodes using constant values is not allowed.

This definition means that as long as the type is defined over the whole group then it is a multiparty communication type even if only a subset of the nodes actually exchange messages. It is the programmer's view that is important for defining the communication types and not the actual exchange of messages. Even if the actual message exchange is between only two nodes, as long as those two nodes are not specifically identified in the type definition (so we can view the communication as an operation over the whole set of nodes), then it is a multiparty communication type. Again, Anycast is the example. We view Anycast as a node sending a message to any other node in the group. So it is a definition over the whole group. However, when we take an instance of the communication type then the "real" nodes are assigned specifics roles.

4. The Programming Model

The programming model is based on a set of independent processes running on different nodes. For the sake of simplicity lets assume a single process per node.

There are two levels of programming.

1. **Type Programming:** Type programming refers to the task of implementing the communication type. That is the task of realizing or implementing a given type as a software entity in a given environment. Type programmers get the specification of the communication type defined in terms of the type system. Once the communication type is implemented it is published together with the formal definition of the communication type. The communication type needed by an application is also defined in terms of the type system. When selecting an implementation of the communication type logical equivalence rules can be used to compare the required type with the implemented types.

   The same communication type can have different implementations. For example, all of the application level multicast schemes mentioned earlier are implementations of the multicast communication type. As different multicast implementations have different goals (optimize bandwidth, reduce latency, etc.), we expect many implementations of the same type that have different optimization goals.

   In our programming model a communication type is implemented as a *class* (no language specific notion is intended). The class captures the details of the communication. For example, a class that implements multicast has the logic to construct and maintain a multicast tree. An instance of a class that implements a communication type is called an *agent*. We discuss the role of the agents in detail in Section 5.3.

   There are, potentially, many different classes for the same communication type. Each class of the same type supports the same communication pattern, but may provide optimizations for different aspects of the communication. Therefore, use of an instance(s) of any of these classes for the communication provides the correct communication pattern, but may not be optimized to achieve the application specific goals (note that all the nodes must agree on one particular implementation).

2. **Application Programming:** The application programmers are shielded from the implementation details of the communication type, which are captured in a class. They deal with the instances of the communication types. An instance of a communication type is called a *session*. To join a session an application needs an agent (of the same communication type as the session) in its node. The application participates in the communication through the interface provided by the agent. The application needs an agent per session—even for two sessions of the same communication type, two agents are needed.

   A session is created by creating the very first agent of the session. This initial agent serves as the bootstrapping point for the session and we call it the *session leader*. Each session has a unique identifier. An application uses this identifier to join the session. This is analogous to the use of a multicast address to join a multicast session. The method used to publish the session identifier and the process that an application uses to join a session is implementation specific. We discuss one particular method to join the session in the next section on the middleware.

In this model a single node may participate in multiple instances of the same communication type or multiple communication types and also the same set of nodes may participate in multiple communication types, all through different sessions. The set of nodes associated with a session is not static and may change over time. Messages are sent and received in the context of the session. The operations of two different sessions are isolated.

5. The Middleware System

In this section we present a middleware to support the implementation and the deployment of multiparty communication types. Note that we use the same name, MayaJala, to identify both the framework and the middleware.

The design of the middleware is based on the above mentioned programming model and an important design decision to represent a session (internally in the middleware) by a virtual overlay network connecting the agents of the session. The decision to use a virtual overlay network for a session is based on the existing works on different multiparty communication types. We note that these works either require special support from the network or build their own overlay networks at the application level to provide that support. There are several overlay network based solutions to multicast [9, 21, 14], Anycast and Manycast [6]. LAM/MPI [4], a public domain implementation of MPI, also constructs overlay trees across the LAM nodes to implement the collective communication primitives, such as `MPI_Bcast()`. Furthermore, Badrinath et al. [2] identify Gathercast as a special case of active networks and Chae et al. [7] discuss an active network implementation of PAMCast. The Concast implementation as described by Calvert et al. [5] requires Concast capable routers in the network—in other words it needs special support from the network. In our design the agents, which are the virtual nodes on the overlay, can provide such "active" functionality.

Based on the above design decision, MayaJala middleware provides the basic functionality to build overlay networks. The type programmers use this functionality to implement the types. The agents use this functionality to maintain the session.

We design MayaJala on the following assumptions:

- We assume that all the nodes are on the Internet. We only assume point-to-point communication capability between all pairs of nodes. We do not require special capabilities, such as the IP multicast capability.
- We do not assume any central control over these nodes.
- The tasks carried out by each node are independent. We do not assume that the nodes perform any cohesive task. From the nodes participating in a session, the only common interest that we assume is the communication.
- All the nodes that join a session, if necessary, forward messages to other nodes in the same session. We do not discuss the mechanism of enforcing this assumption.

We describe the design of MayaJala below. We implement the system in Java, but we try to keep the discussion of the system independent of the implementation language as much as possible. First, we give an overall view of the system and then discuss each component in detail.

5.1. A Bird's Eye View

For the following discussion we assume a single application and a single instance of MayaJala per node. There are other possibilities and we mention some of them in Section 5.5.

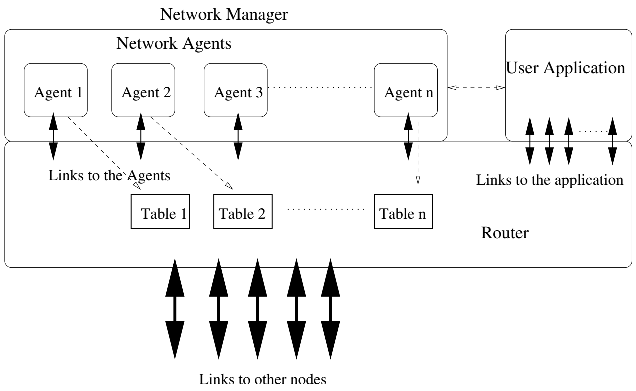

Figure 1. Architecture of MayaJala

The architecture of MayaJala, in one node, is shown in Figure 1. MayaJala consists of four main components.

1. Network Manager
2. General purpose packet forwarding router
3. Agents
4. User application

First we give a bird's eye view of the system to explain the overall operation and then we describe each component in detail.

To join a session the application first needs a reference to a MayaJala instance in its node. The application gives the session identifier to the *network-manager* of the MayaJala instance. The network-manager uses this identifier to download and install an *agent*, which represents the session in the local node. The agent, via the router, installs links from the local node to some selected nodes to join the virtual network of the session. It also installs its own routing table in the router. The router has multiple routing tables—one per agent. The *router* selects a correct table for each incoming packet and forwards packets according to the rules in the table.

We describe each component of the system in detail in the following sections.

5.2. The Router

The router provides the message forwarding mechanism for the virtual networks. The actual routing functionality is a part of the agent and the router only provides the forwarding mechanism and the monitoring functionality. The router is oblivious to the special functions of the individual networks and forwards packets by following the rules in the set of routing tables.

A separate routing table is maintained for each network. The router simply follows the forwarding instructions in the routing table corresponding to the network on which the packet was received. We describe how the routing tables are created and updated in Section 5.3.

The routing tables are identified by the session identifier. The session identifiers are globally unique and there is a virtual network per session. Therefore, the session identifier uniquely identifies a network. The router maintains a set of virtual links from itself to other MayaJala instances, to the application, and to the agents. The router hides the details of the establishment and the maintenance of these links. An agent, when it is installed in the local MayaJala instance, requests the router to establish the necessary links to set up its network. Links are shared between networks, but the agents are oblivious to this fact.

Each packet carries the session identifier in its header. This serves as an index to the correct routing table for the network. The packet header also has information on the *source-address*, *destination-address* and the *message-type* (control or data). Since we are dealing with multiparty communication types, we expect most of the networks not to rely on the *addresses*. For example, in multicast a message is sent to all the nodes in the network and not to a particular node. The topology of the network, in this case the multicast tree, determines the forwarding rules. However, we do not expect this to be the case for all the networks.

There is also a sub-module (not shown in Figure 1) in the router that monitors links that connect the remote nodes. The link-monitor monitors the "liveness" of the links and also measures the link parameters (e.g., bandwidth, latency, and packet loss). The agents use this information to take decisions on routing and also on selecting links for the networks.

5.3. Agents

An agent is the representative of a session in a given node. The agent knows how to join the session and how to form the network and is responsible for maintaining the routing table of the network in the router.

Agents are dynamically installed into the MayaJala instance as needed. It is important to describe the process of creating and installing an agent. We first describe how a communication type is implemented and then go on to describe the installation of an agent in MayaJala.

A communication type is represented by a class. It describes the functionality common to all the instances of the type. It is the task of the type programmer to design and implement the class. The type programmer uses the functionality provided by the MayaJala middleware system to implement the class for the type. The main functionality of the agent is to maintain the virtual network for the session. When codifying this functionality the type programmer assumes the functionality provided by the routers and codes the routing logic (or the policies of using the mechanisms provided by the router) into the class. The type programmer also depends on the functionality provided by the network-manager (described later) to instantiate the agent in the MayaJala instance of a node. The class provides an interface to be used by the network-manager to feed the local information into the agent.

An instance of a communication type, a session, is created by creating an instance of the class, the agent. This initial agent—the session leader—of the session gets a globally unique identifier. MayaJala generates this unique identifier by concatenating the IP address of the node that created the session with a locally unique identifier. This combination is globally unique.

An application that wishes to join a session must have the session identifier. We do not describe the method of publishing the session identifiers. The application gives the session identifier to the network-manager. The network-manager uses the session identifier to locate and download the agent into the MayaJala system. The agents are designed as serializable objects that can be downloaded and installed dynamically. The agents go through another initialization process at the MayaJala system that downloaded it. All the agents provide a general interface to the MayaJala system and the network-manager uses this interface to initialize the agent.

Once installed on the local node the agent also acts as a routing daemon for the network of the session. The agents work above the routing module. This is somewhat similar to the routing protocol daemons in UNIX systems. For example, OSPF or RIP protocol daemons run in the application layer while the actual routing takes place in the kernel. The routing daemon changes the routing table in response to network changes. We follow a similar model for our network agents.

On instantiation, the agent installs a duplex link from itself to the router. The agent also creates its own routing table in the router. It may also ask the router to create links to other nodes and also to the application. The agent knows how to create the network for the session (agents have the logic to contact the corresponding agents on the other nodes and construct the network). It may need information about the links for this purpose and gets the link information from the router. Depending on this information, the agent sets up the routing table and from time to time may update it.

Rather than letting the router to simply forward the messages, an agent may decide to take a more active role in the forwarding process. For example, take the duplication suppression functionality in Concast [5]. Assume that the network for a communication type similar to Concast has a tree topology. All the messages are destined toward the root of the tree. An agent on this network may want to inspect the messages coming through the downstream links of the tree to suppress the duplicates before forwarding them toward the root. To achieve this, the agent inserts rules similar to the following into its routing table.

- Forward packets coming through the downstream links to the agent.
- Forward packets coming through the agent's link on the upstream link.

The router is not aware of the functionality of the agent and it simply forwards packets according to these rules.

An example agent: We use the following example to describe the functionality of an agent. We give an example implementation of the best effort multicast type. First, we describe a multicast tree construction algorithm and then we describe how the agent implements it. We do not claim that the tree construction algorithm is novel and only use it as an example. We also do not discuss the issues such as the robustness or other optimization issues of this algorithm. This basic algorithm is sufficient to describe the operation of the agent.

- The root node is the first node on the tree and it starts the session.
- Any node that wishes to receive messages on the multicast session sends a request to the root. The root sends that request downstream on the tree.
- Each node that receives the request, with some probability p, decides to contact the new node. If there are N nodes currently in the tree, on average Np nodes contact the new node. There is a non zero probability that not even a single node contacts the new node. Therefore, after a timeout the new node sends the request again to the root. The root itself becomes the parent of the first few new nodes (say, first k nodes).
- The new node evaluates the links—according to some measure, such as latency—to all the nodes that contacted it and selects one node to be its parent.

The type programmers program this algorithm into a class. To start a session a node creates an instance of this class, an agent, and publishes its identifier. This initial agent takes the role of the root node in the tree and it is the session leader. On instantiation the session leader gets a link from itself to the router. The session leader takes the following actions.

- It creates a new routing table in its MayaJala instance and inserts a rule in the routing table for each type of message that it expects to get. There are five types of messages.
  - `join_session`: A new node sends this message to join the session and to get the agent.
  - `join_tree`: After initializing the agent, the new node sends this message to the root.
  - `app_data`: A message from the application on the root node.
  - `parent_ready`: A message from a node to another node indicating that it is willing to be the parent of that node in the tree.
  - `child_ready`: A node sends this message to another node, that agreed to be its parent, to indicate the acceptance of that offer.

  At the beginning of the session the routing table has the following entries:

  - Forward `join_session` messages to the agent.
  - Forward `join_tree` messages to the agent.
  - Forward `child_ready` messages to the agent.
  - Forward `parent_ready` messages from the agent to the child (the child node is identified in the message header).

  The `join_session` messages do not come along the links of the network, because the node that sent the request is not yet on the virtual network. That node has sent the `join_session` message to get the agent and without the agent it does not know how to be part of the network. To solve this chicken and egg problem the router listens on a special port for such requests. However, the session leader must still add the join_session rule to the routing table to tell the router where to send the message.

- On the receipt of a join_session message, the session leader creates a clone of itself and sends it to the new node. This is the agent for the new node and it is initialized to act as a non-root node.

- On the receipt of a join_tree message (up to first k such messages), the session leader sends a parent_ready message to the agent that sent the join_tree message. The new node accepts a parent_ready message from the session leader and sends a child_ready message to the session leader. On the receipt of a child_ready message the session leader adds the following rule into the routing table.

  – Forward app_data coming from the application to the new child node.

- After the first k new nodes, the session leader deletes the join_tree, child_ready, and parent_ready rules in its routing table and inserts the following rule into the routing table.

  – Forward join_tree messages to the child nodes of the tree for this session.

On a non-root node the application gives the session identifier to the network-manager to join the session and the network-manager contacts the session leader and gets the agent—this process is described in detail later. Once the agent is instantiated it asks the router to create a link to the session leader. This link is temporary and it will be shut down after the node has joined the network. It also creates its routing table in the router and adds the following entry.

- Forward all the messages to the agent.

The agent sends the join_tree message to the session-leader. As described before the session leader sends this message down the tree or replies directly. Assume that it sends the message down the tree and some nodes decided to be the parent of the tree. A node that is already in the tree has the following rule in the routing table.

- Forward join_tree messages along the downstream links of the tree and also send a copy to the agent.

On receipt of a join_tree message, an agent already in the tree decides with probability p to be the parent of the new node. If it decided to be the parent, then the agent asks the router to create a link to the new node and enters the following entry into the routing table.

- Forward child_ready messages coming through that link to the agent.

Then it sends a parent_ready message to the new node.[3]

The new node may get parent_ready messages from several nodes and it queries the router about the conditions of the links to the nodes that sent the messages. It selects one node to be its parent and asks the router to shutdown all the other links. The router coordinates with other routers to shutdown the links. The node then sends a child_ready message to the selected parent and asks the router to delete the previous entries and inserts the following entries. Note that the new node is a leaf node and does not yet have child nodes.

- Forward app_data messages to the application.

- Forward join_tree messages to the agent.

On receipt of the child_ready message the parent node also instructs the router to forward both the app_data and the join_tree messages to the new node.

Once this is done the agent is not on the critical path of the application data (the data that is multicast by the application on the root node). It has to take an action only when it gets a join_tree message.

5.4. Network Manager

The main functionality of the network-manager is to locate, download and install an agent for a given session. The application gives the session identifier to the network-manager and the network-manager performs the above tasks. The network-manager knows how to interpret the session identifier and locate the session leader to get the agent. Once the agent is downloaded into the local Maya-Jala system the network-manager instantiates the agent by giving it the information about the local node and the constraints on local resources. This information may include the CPU power, type of the connection to the Internet, constraints on the number of links that an agent could create and the amount of bandwidth that an agent could consume. The use of this information is up to the agent. For example, the above mentioned agent could use the information on the constraints on the bandwidth, number of links, and type of

[3]Note that this needs another routing entry as agents do not send messages directly to other agents and only send them via the router.

connection to the Internet to decides on the maximum number of children that it wants in the tree and hence to decide on the probability p.

The network-manager also asks the router to create a local link from the router and passes a handle to one end of this link back to the application. It also gives the identifier of this link to the agent. The agent uses this link identifier to insert entries such as "forward messages to the application", into the routing table—the application is identified by the identifier of the link to the application in such a rule.

The network-manager can monitor the agents resource usage and may take actions to stop runaway agents that violate their resource constraints.

The network-manager hides the complexity of this process from the application. To join a session an application only needs the session identifier and does not have to know about the process of installing the agent or the agent itself.

5.5. User Applications

First, the user application must get a reference to the MayaJala system. There are two options for this. First, we could make MayaJala a part of the application and the application simply creates an instance of MayaJala. The second option is to create a common MayaJala instance for all the applications in a node. There are advantages and disadvantages of each of these two options. Since multiple applications running on a node are expected to have more diverse communication patterns than a single application, the first option utilizes the MayaJala instance better. On the other hand, such an approach increases the complexity of MayaJala because now it has to handle resource management across applications. For the discussion below we assume that the application somehow has a reference to the MayaJala system.

The application gives the session identifier to the network-manager to join a session. On successful instantiation of the agent the application gets a handle to a link. The application uses a known interface (compatible with the agent) to send and receive messages on the session. It passes this handle to the link to the interface to send and receive messages. For instance, in the above mentioned example the router copies `app_data` to the application's link. The interface reads this data and passes them to the application.

The application does not know and does not care about the network. Its view of the communication is through a well defined interface and communication type. The network is just an implementation of the communication type and it is known only to the middleware system. The existence of the network is transparent to the application.

6. Conclusions

We presented a framework, MayaJala, for multiparty communication types. MayaJala has three components; a type system, a programming model, and a middleware system. These three components provide support for different phases, as described in Section 1, of the development of a distributed application.

At the design phase our type system can be employed to describe the communication without committing to any implementation or any interface. The type system defines the communication pattern precisely. The type system also allows for the identification of the communication patterns in the most natural way to the application. It does not force the programmers to think only in terms of the well known communication types. Once the required communication pattern is defined, mathematical reasoning can be applied to the definition of the pattern to investigate whether it is described by a well known (or any already implemented) type. The type programmers advertise the communication type, described in terms of the type system, with the implementation of the communication type. The application programmers can select an implementation by comparing the advertised types with the definition of the required communication pattern. The application can be linked with the different implementations of the same communication type to take advantage of the features of different implementations.

The programming model describes how to implement the communication types and how to use them in applications, thus providing a model for the implementation phase. The programming model describes the abstractions that shield the application programmer from the details of the implementation of the communication type.

The middleware component of the framework provides support for type programming, application programming, and deployment of the application. It provides the functionality needed for the implementation of the communication types. The middleware also provides an environment to install different communication types and shields the application from the details.

MayaJala does not provide any direct support for the debugging of the applications. However, we cited several works that use communication patterns to debug distributed applications. We expect that the use of communication types explicitly in the application, as promoted by MayaJala, would enhance such debugging strategies.

At the moment we are in the process of implementing the middleware component of MayaJala as a proof of concept. The implementation will reveal the practical problems and the performance results. One of the practical problems we are facing now is to determine a good interface of a type to the application. We expect the experience gained by implementing different communication types will help us to determine an interface to a communication type that would be compatible with the expectations of a large set of applications.

References

[1] M. 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.

[2] B. R. Badrinath and P. Sudame. Gathercast: The Design and Implementation of a Programmable Aggregation Mechanism for the Internet. In *Proc. IEEE International Conference on Computer Communications and Networks (IC-CCN)*, pages 206–213, Oct. 2000.

[3] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and F. 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.

[4] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994.

[5] K. L. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: Design and Implementation of an Active Network Service. *IEEE Journal on Selected Area in Communications (J SAC)*, 19:426–437, Mar. 2001.

[6] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *Proceedings of the Fifth International Workshop on Networked Group Communications (NGC'03)*, Sept. 2003.

[7] Y. Chae, E. W. Zegura, and H. Delalic. PAMcast: Programmable Any-Multicast for Scalable Message Delivery. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 25–36, June 2002.

[8] S. Y. Cheung and A. 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.

[9] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of the ACM SIGMETRICS*. ACM, June 2000.

[10] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) specification, Dec. 1998.

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

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

[13] M. Fomenkov, K. Keys, D. Moore, and K. Claffy. Longitudinal study of internet traffic from 1998-2001: a view from 20 high performance sites. Technical report, Cooperative Association for Internet Data Analysis - CAIDA, Apr. 2003.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 197–212, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.

[15] D. Johnson and S. Deering. RFC 2526: Reserved IPv6 Subnet Anycast Addresses, Mar. 1999.

[16] T. Kunz and M. 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, Nov. 1997.

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

[18] 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.

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

[20] J. Pedersen and A. 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.

[21] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS-01)*, pages 49–60, Berkeley, CA, Mar. 26–28 2001. The USENIX Association.

[22] S. Saroiu, P. K. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of internet content delivery systems. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 315–328, New York, Dec. 9–11 2002. ACM Press.

[23] J. Smed, T. Kaukoranta, and H. Hakonen. A review on networking and multiplayer computer games. Technical Report TUCS Technical Report No 454, Turku Centre for Computer Science, Turku, Finland, Apr. 2002.

[24] J. Yoon, A. Bestavros, and I. Matta. SomeCast: A Paradigm for Real-Time Adaptive Reliable Multicast. In *Proceedings of RTAS'2000: The IEEE Real-Time Technology and Applications Symposium*, pages 101–110, Washington, DC, May 2000.

[25] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. 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.