A Framework for Multiparty Communication Types

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

Several multiparty communication paradigms, such as multicast and anycast, have been discussed in the literature and some of them have been used to build applications. There is a vast design space to be explored in implementing these communication paradigms over the wide area Internet. 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.

We provide a framework to address the above problems. The framework includes a model to name the communication paradigms through the notion of **communication types**. *It also provides an API suitable for all communication types. The framework also includes a middleware that facilitates the implementation and deployment of communication types.*

We have implemented a wide assortment of communication types and we demonstrate their utility and the effectiveness of the framework through some simple example applications. We also show that the cost interposed by the middleware is minimal and that the framework facilitates the concise implementation of communication types.

1 Introduction

In addition to the well known communication paradigm multicast, several other communication paradigms that are suitable for distributed applications have been discussed in the literature. Anycast [7, 31], Manycast [13], Concast [12], CollectCast [20], and SomeCast [36] are some of them. On close inspection it is clear that even the term multicast refers to a family of communication paradigms rather than a single well defined communication paradigm. The members of this family differ in delivery guarantees, ordering guarantees, and the constraints placed on senders and receivers. For example, the multicast paradigm proposed for IPv4 [17] only provides best effort delivery without any ordering guarantees and allows any process, including those who are not in the group, to send messages. In contrast, the multicast paradigms ABCAST and CBCAST [10] provide strict ordering and delivery guarantees. The multicast paradigm implemented by Overcast [22] and Express multicast [21] are examples of single-source multicast.

The other communication paradigms have not been studied as extensively as multicast. There have been proposals for anycast for IPv4 [31] and it is included in IPv6 [23]. Anycast has also been implemented in application-level overlay networks [13] and by changing the DNS name resolving process at the application level [7]. These implementations provide different semantics. The paradigms such as Concast and SomeCast have only been discussed with one particular implementation. However, it is easy to see that there are possible variations of these communication paradigms as with the case of multicast.

There is a vast design space to be explored in implementing these communication paradigms over the wide area Internet. The large number of end-system multicast implementations are evidences of this design space. These implementations explore ways to maximize bandwidth; minimize latency, link stress and stretch [16]; 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 [26].

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 [28, 29] and techniques such as Script [19] 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 common 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 other 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 provide a framework to overcome the above mentioned obstacles and to allow applications to use communication paradigms independent of their implementations. This includes an abstract model that allows a large number of communication paradigms to be precisely defined; we call these *multiparty communication types*. 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.

We have implemented several communication types using this middleware. We demonstrate the flexibility that this system gives to the programmers through a simple application that uses two communication types simultaneously for its operation. We demonstrate that the middleware allows communication types to be implemented concisely by comparing our implementations of communication types against the Macedon [34] implementations of overlay networks with similar functionality ; Macedon is considered as providing a concise language to build overlay networks. We also show that the cost incurred by the middleware itself is minimal.

2 Backgound

Multicast, even though it has been included in IPv4, is not widely deployed in the global Internet; it has limited deployment through the Mbone. There are several implementations of IP multicast, in terms of different routing protocols such as DVMPRP, PIM, and CBT. To the end user these routing protocols are transparent; these protocols are the evidence of the large design space of IP multicast. On the other hand Express multicast [21] changes the semantics by restricting the sources to a single source; this allows Express to scale to large set of receivers. In RFC3569 [8] Bhattacharyya et al. describe regular IP multicast as Any-source multicast and also mention Source-Specific multicast and Source-Filtered multicast.

In contrast to IP multicast, application-layer multicast (or end-system multicast) does not depend on router support and implements multicast by forming application-level overlay networks. There are several end-system multicast implementations such as SMOP [5], Overcast [22], Narada [16], and ALMI [32]. These implementations differ in overlay construction techniques, optimization objectives, message semantics, and APIs.

Birman et al. [10] introduced several multicast primitives with different delivery and ordering guarantees for group communication. The Spread system [1] provides several multicast semantics implemented over the wide area Internet. The Ensemble [9] system provides a framework to compose different group communication protocols (multicast family) by stacking a set of micro-protocols. This system allows different implementations of the protocols to be used transparent to the application.

Although there has been an RFC [31] proposing anycast for IPv4, it has never been implemented. To address the

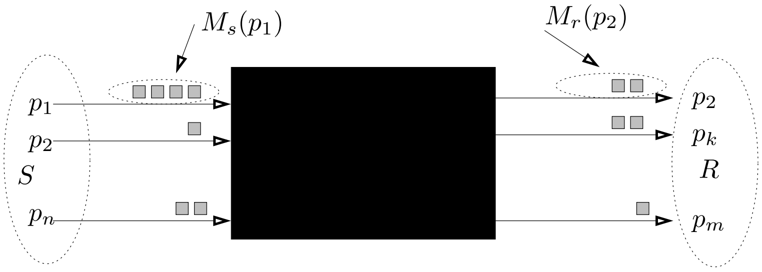

Figure 1: The black box model of multiparty communication.

scalability issues of IP anycast, Katabi et al. [25] present an alternative architecture of IP anycast. Anycast proposed for IPv6 is vastly different from the one proposed for IPv4. The application-level implementation of anycast by Castro et al. [13] is also vastly different from the implementation by Bhattacharjee et al. [7] in implementation details, API, and message semantics. The implementation by Castro et al. [13] also provides Manycast—the paradigm of sending a message to several destinations.

Calvert et al. [12] describe Concast as the inverse of multicast—messages sent by multiple sources are delivered as a single message to a single destination. Concast uses a single address to identify the group of senders. Gathercast [3] is another multipoint-to-point aggregation service. Gathercast hides the fact that packets are aggregated while Concast exposes this to the application.

Chae et al. [14] 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. Cheung et al. [15] introduced a similar communication pattern, Quorumcast, almost a decade before the above works. Quorumcast was introduced as a communication paradigm akin to quorum consensus synchronization. Probabilistic multicast [2] also has a similar communication pattern, but the set of nodes to receive the message is determined probabilistically.

Yoon et al. [36] 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.

MPI [28, 29] 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 [28]) 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 [29] 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. 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 [18]. 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.

3 Communication Types

In the previous sections we highlighted the fact that names such as multicast do not identify unique communication paradigms. We develop a model that allows us to name different communication paradigms precisely through the notion of communication types. We also use this model to identify a common API that can accommodate all the communication types and a design strategy for the middleware.

We model multiparty communication as a black box and a set of processes, P, connected to the black box (Figure 1). Each process sends messages to and receives messages from the black box. We model the messages sent into the black box by a sequence, $M_s(p)$, associated with each process $p \in P$. Similarly, we model the messages coming out of the black box to a process $p \in P$ by the sequence $M_r(p)$. Each element of $M_s(p)$ is globally unique. Each process p can be *send enabled* in which case $p \in S$ where S is the set of send enabled processes. Similarly the *receive enabled* processes are in the set R. A process p can be in both S and R. There is also a number, $N(p)$, associated with each process p.

This number is constant and does not change over time. A process p changing its membership in either of the sets S or R or changing the number is modeled as p simply stopping sending and receiving messages and re-joining P as a new process. The size of P monotonically increases and P retains the whole history of the messages sent and received. The sets S and R can grow and shrink.

This is a dynamic process and to describe it we take a snapshot of the process. That is, we stop processes from sending anymore messages into the black box and let the black box deliver, if at all, all the messages to be delivered to the receivers. We can take a snapshot of the system after any external action, which includes sending a message and any change in S, R, or P. All the snapshots must respect the causality property; that is, a still picture that has a message in one of the receive sequence but that message does not appear in any of the send sequences is not a snapshot.

The black box expects S, R, and P to have certain properties at each possible snapshot. We call this property the *port protocol* of the black box. The port protocol is expressed as a predicate on S, R, and P. This predicate defines the properties expected of S, R, and P and the relationship between S, R, and P. The port protocol does not refer to the message sequences associated with the processes. It is up to the processes collectively or some external entity that has control over the processes to ensure that the port protocol is followed at all possible snapshots.

If the port protocol 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 protocol* of the black box. The message protocol is expressed as a predicate on the set P. The complete history of the send and receive sequences are available through P and the message protocol 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 do not show the complete history of the messages sent into the black box; there could be processes that were in the set S in a previous snap shot and that have sent messages to the black box. Same is true for the set R. The processes follow the port protocol and expect the black box to follow the message protocol. The black box expects the processes to follow the port protocol and guarantees the message protocol as long as the port protocol is followed.

Taken together, the port protocol and the message protocol completely define the operation of the black box. If we replace a black box by another that expects the same port protocol and follows the same message protocol 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 protocols are of the same *communication type*. A communication type t is defined by the tuple $< P_t(S, R, P), M_t(P) >$ where $P_t(S, R, P)$ is a predicate that defines the port protocol and $M_t(P)$ is a predicate on the set of processes P that defines the message protocol.

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 life time of the multicast session and the source does not receive the messages. We denote this communication type by $ssmcast$ and define it as follows.

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

We use the symbol $\sqsubseteq$ to denote a subsequence where $x \sqsubseteq y$ indicates that x contains only the elements in y in the same order but may not contain all the elements of y. Also in the communication type definitions 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.

The port protocol, $P_{ssmcast}(S, R, P)$, requires $|S| = 1$ on every snapshot. This also 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 protocol since there is a snapshot that makes $S = \emptyset$. In the message protocol we do not have to specify that the send sequences on the receivers are empty. This is implied in the port protocol. Since $S \cap R = \emptyset$ the send sequences on the receivers must be empty and we do not have to explicitly include this in the message protocol. The message protocol must be understood in the context of the port protocol. 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 protocol. 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. The set P keeps the whole history of the messages sent and received. Therefore, we only use P in defining the message protocol.

Note that $ssmcast$ is just a nickname for the above communication type. The actual name is given by the tuple $< P_{ssmcast}(S, R, P), M_{ssmcast}(P) >$. 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 paper 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.

$$P_{atoa}(S, R, P) : \ \mathbf{true}.$$
$$M_{atoa}(P) : \ \forall_{p \in P} \ \exists_{x \in SendInterleave(P)} \ [M_r(p) \sqsubseteq x].$$

Where,

$$SendInterleave(P) : \{y : (\forall_{s \in P} \ [M_s(s) \sqsubseteq y]) \wedge (\forall_{z \in y} \ \exists_{p \in P} \ [z \in M_s(p)])\}.$$

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 protocol does not impose any restriction.

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.

$$P_{anycast}(S, R, P) : \ |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$
$$M_{anycast}(P) : \ \forall_{p \in P} \ \forall_{m \in M_s(p)} \ \exists_{q \in P} \ [\ m \in M_r(q) \ ].$$

Since the message protocol has to be true in each possible snapshot this means that each message must be delivered before the next message. A message protocol that guarantees message delivery also means in-order delivery. This is a limitation of the model and we discuss this further in Section 5.

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.

$$P_{anycast\_unique}(S, R, P) : \ |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$
$$M_{anycast\_unique}(P) : \ \forall_{p \in P} \ \forall_{m \in M_s(p)} \ \exists!_{q \in P} \ [m \in M_r(q)].$$

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

$$P_{anycast\_amo}(S, R, P) : \ |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$
$$M_{anycast\_amo}(P) : \ \forall_{p \in P} \ \forall_{m \in M_s(p)} \ [(\exists!_{q \in P} \ [m \in M_r(q)]) \ \vee (\forall_{r \in P} \ [m \notin M_r(r) \ ]) \ ].$$

There is no ordering guarantee in $anycast\_amo$.

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

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. The message is received at exactly N receivers and there is a set of senders distinct from the set of receivers. There are no ordering guarantees.

$$P_{manycast}(S, R, P) : \ (\forall_{s \in S} \ [N(s) \leq |R|]) \wedge \ S \cap R = \emptyset.$$
$$M_{manycast}(P) : \ \forall_{p \in P} \ \forall_{m \in M_s(p)} \ \exists!_{Q \subset P} \ [\ |Q| = N(p) \ \wedge \ \forall_{q \in Q} \ [m \in M_r(q)] \ ].$$

$manycast$ 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 $manycast$. N indirectly gives a meaning to a message and the black box is aware of N and must act based on N. This somewhat breaks the notion that the messages are meaningless to the black box. Later in this chapter when we discuss type conformance we show that N can cause problems identifying conforming communication types. An unattractive alternative to using N is to define large number of communication types such as 1-cast, 2-cast etc. This solution is unattractive in practice because a process communicating over a $manycast$ session may want to change the intended number of receivers at runtime. If we fix the n in the n-cast by selecting one particular n-cast this is not feasible. Our current definition allows different processes to indicate different number of recipients on the same session of the communication types. Another solution is to define a communication type that takes $N(p)$ as a parameter. This solution too has the similar problems as the previous one and in addition do not solve the type conformance problem either.

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. All the messages are received at the single recipient. The single recipient does not send any messages and the senders do not receive messages.

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

$gather$ can be considered as the inverse of the $ssmcast$ where messages from a single source are received by all the other processes. 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 protocol since this is implied by the port protocol. Even if this fact is explicitly included in the message protocol it defines the same communication type since the message protocol must be interpreted in the context of the port protocol. However, omission of such details results in a concise definition of the message protocol.

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

$$P_{somecast}(S, R, P): \ \forall_{r \in R} [N(r) \leq |S| \ \wedge \ S \cap R = \emptyset].$$
$$M_{somecast}(P): \ \forall_{p \in P} \forall_{x \in subs(M_r(p), N(p))} \exists!_{Q \subseteq P} \forall_{m \in x} \exists_{q \in Q} \\ [m \in M_s(q) \ \wedge \ |Q| = N(p) \ \wedge \ \forall_{q_1 \in Q} \exists_{m_1 \in M_s(q_1)} [m_1 \in x] \ ].$$

$subs(s, n)$, where s is a sequence and n is an integer, is a set of all complete subsequences of size n of s.

This definition of $somecast$ requires that each sequence of $N(p)$ messages received at the process p to be received from a different sender. 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 protocol. The communication type $gather\_mcast$, defined below, uses the set P in the definition of its port protocol.

$$P_{gather\_mcast}(S, R, P): \ \exists_{p \in P} [p \in S \ \wedge p \in R \ \wedge \\ \forall_{q \in P - \{p\}} [N(p) \neq N(q) \ \wedge \forall_{r \in P - \{p\}} [N(q) = N(r)]] \ ].$$
$$M_{gather\_mcast}(P): \ \exists_{p \in P} \forall_{q \in P - \{p\}} [N(p) \neq N(q) \ \wedge \\ M_r(p) \sqsubseteq SendInterleave(P - \{p\}) \ \wedge \ M_r(q) \sqsubseteq M_s(p) \ ].$$

The communication type $gather\_mcast$ uses $N(p)$ to identify a unique process. The port protocol 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 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 protocol 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 protocol can be followed. Once more than two processes are in P we can identify the unique process. The port protocol requires that all the processes other than the unique processes to 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.

$$P_{unicast}(S, R, P): \ S = R = P \ \wedge \ |P| = 2.$$
$$M_{unicast}(P): \ \forall_{p \in P} \forall_{q \in P} [p \neq q \Rightarrow M_s(p) = M_r(q)].$$

This definition of $unicast$ requires that all messages are delivered and that they are delivered in-order. The message protocol can easily be changed to define other flavors of the $unicast$ communication type, such as the communication type that does not provide delivery guarantee but still delivers messages in-order and the communication type that does not even provide ordering guarantee. All these types can be defined with the same port protocol.

Note that in the above definition of $unicast$ the port protocol requires that the size of the set P to be 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 protocol requires exactly two processes in P. Both processes must be in S and R in all the snapshots.

3.0.1 More on port protocols

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

$$P_{bogus}(S,R,P) : S \cap R = \emptyset.$$
$$M_{bogus}(P) : \ \forall_{p \in P} \ \forall_{m \in M_s(p)} \ \exists !_{Q \in P} \ [\ |Q| = N(p) \ \wedge \ \forall_{q \subset Q} \ [m \in M_r(q)]\ ].$$

The message protocol of $bogus$ is same as the message protocol of $manycast$. However, the port protocol is different. The port protocol 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. It is impossible to guarantee the message protocol all the time under this port protocol.

In this discussion we simply assumed that the processes follow the port protocol. In practice it is up to the programmers who write the code for these processes to make sure that the port protocol is followed at all times. However, the responsibility of ensuring the port protocol 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 servers 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 to 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 protocol. The idea of the port protocol goes beyond the idea of an interface between programming elements (objects, layers, etc.) because it extends beyond the program text.

3.1 Communication type equivalence

We use first order predicate logic to define the port protocol and the message protocol. Infinitely many different formulae can be used to define the same port protocol and the same message protocol - 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.

$$\forall_{S,R,P} \ P_{t_1}(S,R,P) \Leftrightarrow P_{t_2}(S,R,P) \wedge M_{t_1}(P) \Leftrightarrow M_{t_2}(P).$$

Take for example the communication type x.

$$P_x(S,R,P) : \ |R| \geq 1 \ \wedge \ S \cap R = \emptyset.$$
$$M_x(P) : \ \forall_{p \in P} \ \forall_{m \in M_s(p)} \ \neg(\forall_{q \in P} \ [\ m \notin M_r(q)\ ]).$$

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

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, this processes cannot be automated in general since predicate logic does not constitute a decidable system [6].

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.

3.2 Type conformance

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

$$\forall_{S,R,P}\ P_{t_a}(S,R,P) \Rightarrow P_{t_b}(S,R,P) \tag{1}$$
$$\forall_{S,R,P}\ M_{t_b}(P)/P_{t_a}(S,R,P) \Rightarrow M_{t_a}(P) \tag{2}$$

Where $M_{t_b}(P)/P_{t_a}(S,R,P)$ stands for $M_{t_b}(P)$ in the context of $P_{t_a}(S,R,P)$.

This idea of type conformance was borrowed from Emerald [11, 33]. 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 Section 6 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. $P_{ssmcast}(S,R,P) \Rightarrow P_{atoa}(S,R,P)$ is trivially true since $P_{atoa}(S,R,P)$ is defined as $\mathbf{true}$. In the context of $P_{ssmcast}(S,R,P)$, $SendInterleave(P)$ becomes the singleton set that contains the send sequence of the only sender. Therefore, in the context of $P_{ssmcast}(S,R,P)$, $M_{atoa}(P) \Rightarrow M_{ssmcast}(P)$. 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

$$P_{gather}(S,R,P) \Rightarrow P_{anycast}(S,R,P).$$

Since the only difference between the port protocols 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. We can also show that

$$M_{anycast}(P)/P_{gather}(S,R,P) \Rightarrow M_{gather}(P).$$

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 protocol 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 protocol 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.

Intuitively $manycast$ conforms to $anycast$. However, the port protocol of $manycast$ is defined in terms of $N(p)$ associated with each process p. But the definition of $anycast$ does not use $N(p)$ and $N(p)$ for each process is left undefined. If we had the requirement $\forall_{s \in S} N(s) = 1$ in the port protocol of $anycast$ then we can show that $manycast$ conforms to $anycast$. However, such a condition in $anycast$ port protocol is arbitrary and imposes an unnecessary restriction on the processes.

Note that the direction of the implication of Property 1 is the reverse of the direction of the implication in 2. This is similar to the contravariance property of the Emerald [11, 33] 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 protocols ensures that the port protocol of the type to be replaced can be used with the new type. And Property 2 ensures that under the old port protocol the new message protocol provides the same behavior as the old message protocol.

```
Methods to get the Interface object;

   Interface joinSession(InstanceId id);
   Interface initSession(Agent implementation);

Methods on the Interface object:

   void sendEnable();
   void recvEnable();
   void sendDisable();
   void recvDisable();
   void setParam(int n);
   void leaveSession();
   send(Object o);
   Object recv();
```

Figure 2: The application programmer's interface to communication types.

The idea of type conformance allows an implementation of a communication type to be switched with an implementation of another communication type, not just another implementation of the same communication type. This is possible because of the division of the responsibility between the users and the implementation through the port protocol and the message protocol. As we mentioned before the responsibility of following the port protocol extends beyond the application programmers.

Liu et al. [27] presented the idea of *protocol switching*, which is somewhat similar to our notion of communication type conformance. The switchable protocols are identified with respect to several predefined *meta properties* that are preserved under a particular runtime switching mechanism for the Ensemble system [9]. In contrast, we do not impose such restrictions and communication type conformance is more general.

3.3 Sessions of communication types

The black box guarantees the message protocol only if the processes follow the port protocol. In the abstract model we assume that the port protocol is followed from the very beginning. However, in practice this is not feasible in all cases. Take for example the simple communication type of $unicast$. The port protocol of $unicast$ requires that there are exactly two processes connected to the black box and they both are send and receive enabled. In practice it is impossible for both processes to join the session at the same instant in time. This is especially difficult considering the fact that there may not be a central controller over these processes. Therefore, in practice it is inevitable that sessions of some communication types may not follow the port protocol from the very start. As a solution to this pragmatic problem, we do not consider a session as started until it satisfies the port protocol. An implementation does not have to follow the message protocol until the processes in the session started following the port protocol.

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

4 Application programmer's interface

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 application programmer's interface (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. However, the API is not language specific. The complete API is given in Figure 2.

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

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

`joinSession()` is called by the process that wants be added to the set P and the call returns an `Interface` object. All the other methods are invoked on that `Interface` object. Similarly a process that wants to be removed from the set P calls `leaveSession()` on the `Interface` object that it has already obtained by a previous call to `joinSession()`.

`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 parameter $N(p)$ associated with the process. Whether a process can call these methods depends on the port protocol. 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 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, R, and P 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.

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.

5 Limitations of the model

The model has certain limitations that inhibit it from defining some interesting communication types. A message protocol of a communication type that guarantees message delivery also implies that the messages are delivered in order. The reason for this implication is the requirement that message protocol to be true on each possible snapshot. Take for example the communication type $gather$.

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

$gather$ requires that each message in any send sequence appear in a single receive sequence. We can take a snapshot after sending each message. For the message protocol of $gather$ to be true this message must appear in the receive sequence of the single recipient at that snapshot. Note that the definition of the snapshot allows the black box to deliver messages. This means that the current message must be delivered before the next message. Therefore, we cannot define a $gather$-like communication type without the message ordering guarantee.

Another limitation can be found in communication types that do not guarantee message delivery. Take for example $ssmcast$.

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

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

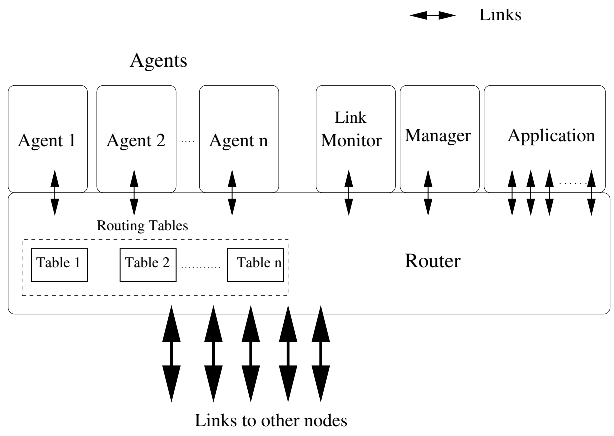

Figure 3: The architecture of MayaJala.

The message protocol 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 protocol is satisfied. This model does not have any facility to define the probability of message delivery or that messages will be delivered eventually. Note that the message protocol of $ssmcast$ has the flexibility to deliver a message sent in one snapshot in a later snapshot without violating the port protocol. The message protocol explicitly requires the messages delivered to be delivered in the order they were sent as in this case the definition of the snapshot does not guarantee that.

It would be convenient if we can define communication types that would deliver messages "eventually". The simple model we presented is not capable of that. It would be an interesting research avenue to augment the model with temporal logic to achieve this.

The model does not disallow definitions of communication types such as $bogus$. This communication type has a port protocol that does not support its message protocol. It would be convenient if such definitions can be weeded out by imposing restrictions on the communication types that can be defined.

We deliberately omitted the use of sets S and R in defining the message protocol since these sets are transient. However, in some cases where we restrict these sets to a constant number they lose this transient property. Take for example, the set S in $ssmcast$. It is not clear at this point whether to include these sets in the definition of the message protocol. We opted to keep the message protocols simpler by not using S and R.

There is room for improvement in the model. At this point it is not clear whether a completely different approach can give us a better model. However, the current model with its limitations have served our purpose of presenting the concept of multiparty communication types.

6 Design and Implementation

In the last section we discussed one component of our framework, a model that defines communication types. In this section we present the design of the middleware component, MayaJala. The goal of MayaJala is to allow application programmers to use many communication types in the same application at the same time without worrying about the implementation details of the communication types. We also want to allow components of the distributed application to be written by different programmers (such as the different Instant Messaging clients), and we also want to shield the decision of using a particular implementation of a communication type from the programmers as much as possible. The application programmers only have to know the communication type and the application component's role in the port protocol. Finally, we want the middleware to provide the common functionality required to implement the communication types.

Figure 3 shows the architecture of MayaJala. There are five main components; the router, the manager, the agents, the link monitor, and the application. We implemented MayaJala in Java. In our implementation each of these components runs as a separate thread. Although we discuss the design with respect to the implementation, the concepts are applicable independent of the implementation language. We discuss the components of MayaJala individually, starting with the application.

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

Figure 4: An example server.

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

Figure 5: An example client.

6.1 The Application

The application in Figure 3 is any arbitrary application that uses MayaJala for communication. The main purpose of this work is to allow such applications to use communication types transparently. We start this section by presenting a very simple example application that uses the communication type $ssmcast$.

Figure 4 shows a multicasting server that repeatedly multicasts a message using the communication type $ssmcast$. `MJSSMAgent()` is an an agent that implements the communication type $ssmcast$. The server first creates a MayaJala instance and calls the `initSession()` method of the MayaJala instance to start a $ssmcast$ session. The `initSession()` method returns an *interface* object which provides the API we discussed in Section 2. The session gets a unique identifier and the server simply prints it to the terminal. Then the server uses the interface object it received to send messages. Note that according to the port protocol of $ssmcast$ there is only one sender and there must always be exactly one sender. Therefore, we must start the sender first and also there is no need to call `sendEnable()` in this case because according to the port protocol this node is by default send enabled.

The client code is given in Figure 5. The client gets the textual form of the `NetId` of the instance of the $ssmcast$ in the command line. It uses that to join the session and receives 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 the communication type. The client only needs to know the identifier of the instance and the port protocol of the communication type. According to the port protocol of $ssmcast$, all nodes, other than the source, can only call `recv()` (since this is the only method possible, the interface is by default receive enabled). The agent for the session is dynamically downloaded and installed on the client node transparently.

To use a different implementation of $ssmcast$ only one line in the server code has to be changed. For example, we could use `RandomTreeMcastAgent` (another agent that implements $ssmcast$) in place of `MJSSMAgent` 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. Our design allows for hot swapping of implementations and also has the facility to allow for change of the implementation of the communication type without changing the

MJNetId	TTL	In_link	Source	Destination	Type	Payload

Figure 6: An `MJData` message.

application at all by adding a level of indirection. However, we have not implemented these features in this version of MayaJala.

This application can also use an implementation of a different communication type, $atoa$ (we have implemented this type with the agent `AlltoAllMcastAgent`). This is possible because $atoa$ conforms to $ssmcast$. That means under the port protocol of $ssmcast$ the behavior of $atoa$ is equal to $ssmcast$. 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* [4] of the application. The property of communication type conformance further increases the option value by allowing applications to use implementations of communication types that conform to communication types already in use.

Note that there is no programming language representation of communication types. The communication type system is another tool, similar to design patterns, available to simplify the design and deployment of distributed applications.

6.2 The Router

The agents, such as the `MJSSMAgent` and the `RandomTreeMcastAgent`, implement communication types as overlay networks. These networks are invisible to the application and the application only sees communication types. The router simplifies the overlay implementation by providing common functionality required to build overlay networks.

The router is connected to the other components of the MayaJala instance (the agents, the manager, and the application) and the other MayaJala instances through a set of links. It gets *messages* along these links. There are three sub types of messages; commands (`MJCommand`), data (`MJData`), and requests (`MJRequest`). Commands are issued by the other local components of the MayaJala middleware to get services from the router. Requests are the control messages exchanged between the routers of different instances of MayaJala. `MJData` messages are used by agents to implement networks and to send data on those networks. Each `MJData` message belongs to one and only one network.

The structure of an `MJData` message is shown in Figure 6. 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 caries a network identifier (`MJNetId`) that uniquely identifies a network; there is a network for each session of a communication type and `MJNetId` 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 returned by the matching function. It is possible to extend the routing classes easily to other link selection functionality, such as selecting the least loaded link. The `MJData` messages can be matched against the rules directly or through a set of optional bit masks defined for each attribute to be matched. As shown in Figure 3 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 and the communication between the MayaJala instances goes through the router.

The forwarding rules are applied only to messages of type `MJData`. The router does not interpret the payloads of messages of type `MJData`. They are passed into the agents, the manager, the application, or along the external links according to the rules in the routing table.

The components running on top of the router use `MJCommand*` messages to request services from the router. For example, a message of the type `MJCommandCreateLink` is sent to the router to request a link to some remote instance of MayaJala. As discussed later the agents are responsible for each network and they use `MJCommandUpdateRoutes` to request the router to update the routing table with a new set of rules. The router also provides a timer service. An agent, the application, or the manager could send a message of the type `MJCommandRemindMe` to the router to request a wakeup call at a given time.

The router hides the implementation details of the links from agents. Agents are given an impression of in-order delivery message-oriented links. The decision to hide the implementation details was taken so that the router implementation can evolve without affecting the agents. Agents only get a link identifier which they use for routing rules. In the current implementation the router only creates a single TCP link between two nodes and multiplexes that transparently between all the agents. However, this implementation can be changed, for example to use SCTP (which provides reliable message oriented protocol by itself) instead of TCP, or to create multiple TCP links between two nodes.

The router also provides facilities to monitor and measure the links. The link monitoring functionality is implemented in the link monitor, which is a private module of the router. The agents use an `MJCommand` message to obtain the link measurements and are not aware of the mechanism of obtaining such measurements.

The router never blocks on any operation. In our implementation it is written as an event driven program with non-blocking input output operations. We use the Java NIO package to implement the router. The agents can only interact with the router using messages and the agents do not have references to the router. We decided to use messages instead of up and down calls to prevent a misbehaving agent affecting other agents and the router. As we discuss later, agents are dynamically loaded at run time and the message interface allows for the agents to be sand-boxed.

6.3. The Agents

As discussed above, the `MJData` messages are routed according to a routing table. However, the router does not create or maintain these routing tables. Maintaining a routing table is the responsibility of an agent. The agents implement the communication types as overlay networks. The functionality of the agents is similar to the Unix routing daemons.

To implement a communication type an agent uses the services provided by the router. An agent uses `MJCommand*` messages to get services from the router. As discussed before it may use `MJCommandCreateLink` to get a link to a remote MayaJala instance. The details of creating a link are transparent to the agent. The agents are typically written as single threaded event driven programs. They respond to the incoming messages and incoming responses to the commands issued to the router. Unlike the router, the agent's message receiving calls are blocking calls. However, the agent could set time outs to the receive loop by sending a `MJCommandRemindMe` message to the router. This timer service could be used to schedule other events by the agent.

The agents use `MJData` messages as a means of sending application data as well as control messages for the network itself. `MJData` is structured in such a way to provide this flexibility. The router uses only the header fields in an `MJData` message to match against the routing rules. 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, destination fields, and the payload for its own purposes. The agent 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 the agent can decide on how the messages are routed in the networks. As far as it is concerned the agent could assume that it is the only network that operates on the router. The agent uses the routing rules to guide `MJData` messages on its network. The agent could inject routing rules that are based on the incoming link, the source, the destinations, and the message type into its own routing table in the router. For example, an Agent that implements single-source multicast may insert a routing rule into the source node's router to forward all the messages coming from the application to downstream nodes in the delivery tree; in non-source nodes the agents could instruct the router to forward messages from the upstream to the application and also to nodes further downstream. This is a very simplified description of the agent's functionality. Usually, the agents are required to handle control messages to maintain the delivery tree and must respond to the failures of nodes and requests to join the network.

As we noted previously there is a large design space in implementing a single communication type and different agents can be implemented for a single communication type with different design objectives. One advantage of our design is that it allows implementors of communication types to explore the design space while letting the router explore the avenues to optimize the individual link capacity and employ different techniques to measure links. We have implemented three different implementations for the communication type $ssmcast$ (single-source multicast). We also implemented agents for $gather$, $atoa$, uni, and a variant of any. The functionality provided by the router has simplified their implementations. Due to space limitation we do not give implementation details.

6.4 The Manager

The manager is responsible for locating and downloading the agents corresponding to the session identifiers and it also acts as a repository for agents. In the above example application the manager on the server node starts the agent and issues a globally unique identifier for the session. The manager also stores a copy of the agent. On the client node the manager contacts a resolver to resolve the session identifier to an agent. In the above example the manager on the client node contacts the manager on the server node since the session identifier encodes that as the resolver for that session. The client manager then downloads the agent and starts it.

The agents do not have any reference to other components in the MayaJala instance and they communicate with the router only through messages. The manager creates a link to the router and gives each agent its link on starting the agent.

The design allows for sandboxing agents by using different class loaders and different security policies. The current version of MayaJala does not implement this feature. In the current version we attempted to provide protection but not security, but by isolating the agent-loading and instantiating functionality in the manager we allow for further development in this area.

The manager is implemented as another agent, but with more privileges. It also creates its own network to communicate with managers on other MayaJala instances and uses the general routing facilities provided by the router to all the agents. The manager is also implemented as an event driven program that responds to messages coming through the router.

6.5 The Link Monitor

Measuring the bandwidth and latency of Internet paths is a specialized task. Bandwidth measurement is an ongoing research area [26, 35]. 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. [30] 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 is a private module of the router and it 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. 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.

7 Evaluation

We evaluate our implementation on several aspects. First, 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. The second aspect that we evaluate is the support provided by the middleware to implement communication types. We evaluate this by implementing two multicast implementations and comparing them with equivalent overlay networks implemented in Macedon. Finally, we evaluate the cost interposed by the middleware itself and we show that this is negligible in comparison to wide area Internet costs.

7.1 Simplicity of the Application Code

To demonstrate the simplicity of the communication code in applications that uses MayaJala we implemented a more complex example than the one we discussed before. This example is a chat application. The code for the chat server is given in Figure 7 and the code listing for the client is given in Figure 8. The server acts as a moderator for the chat session and sanitizes the messages. The client is an application with a GUI and we do not show the GUI code.

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

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

Figure 7: The chat server.

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 protocol 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 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 5 communication related lines of code in the server. The implementation of any of the communication types can be changed without affecting the number of lines of code at all—the number of lines of code in the user application does not depend on the complexity of the implementation of the communication type.

When the chat client starts, it joins the two sessions created by the server. 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 5 communication related lines of code and the change of the implementation of the communication type does not affect the number of lines of code and nor does it results in any change of code. We have tested this application with several implementations of the communication type $ssmcast$.

7.2 Support for Building Agents

Macedon [34] 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 [22] overlay network using MayaJala. We completely 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 in 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

```
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]));
   Client cl = new Client(gather, mcast, args[2]);
   cl.showGUI();
   cl.showDiscussion();
 }
}
```

Figure 8: The chat client.

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 1. Our implementation consists of 289 lines of codes while the Macedon implementation has 311 lines of codes. These numbers are very encouraging specially considering the fact that Macedon is a domain specific language for building overlay networks while our implementations uses 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, grand parent, 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, `RandomTreeMcastAgent`, 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

	MayaJala	Macedon
Overcast	289	311
Random Tree	82	142

Table 1: The number of lines of code.

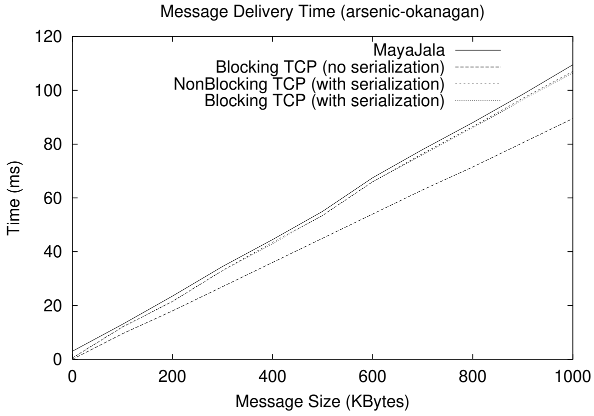

Figure 9: Message delivery time.

use it for comparison.

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 requests to the root node. The agent, `RandomTreeMcastAgent`, 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 also given in Table 1. `RandomTreeMcastAgent.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.

7.3 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 2.6.4-52-smp and okanagan runs kernel version 2.6.5-7.111-smp. arsenic is a dual processor (2.40 GHz Intel Xeon) machine with 512KB with 1GB of memory. okanagan is a quad processor (2.66 GHz Intel Xeon) 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 a echoing back messages sent by okanagan from arsenic. The results are given in the graph in Figure 9.

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 10 shows the results of the same experiment for small messages in the range of 0 to 2 KBytes. The drop in

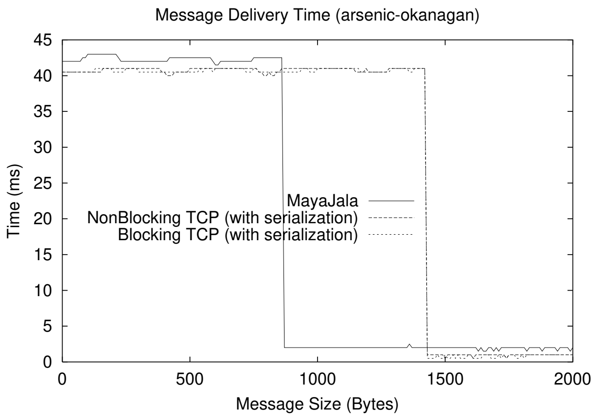

Figure 10: Message delivery time (small messages).

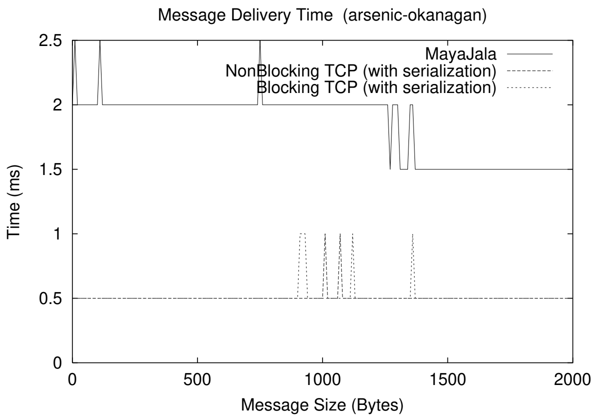

Figure 11: Message delivery time (with Nagle dissabled).

message delivery time around 900 bytes for MayaJala and around 1400 bytes for other two cases is due to the Nagle algorithm. Figure 11 shows the message delivery time with Nagle algorithm disabled. An 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 10. When the Nagle algorithm is in effect this overhead works for MayaJala rather than against it. Figure 10 highlights the fact that there are artifacts, such as the Nagle algorithm, that we take for granted and have far greater impact on the performance than the overhead of MayaJala.

We also tested MayaJala over two nodes on the Emulab testbed and the results are shown in Figure 12. 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 the 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. A routing table that contains one routing rule (with one output link) needs 538 bytes and each additional routing rule adds 83 bytes. It takes less than 1 ms to send an update of a routing table that contains 10 routing rules to the router.

Note that when building MayaJala we chose fast prototyping against 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 build a prototype with minimal overhead.

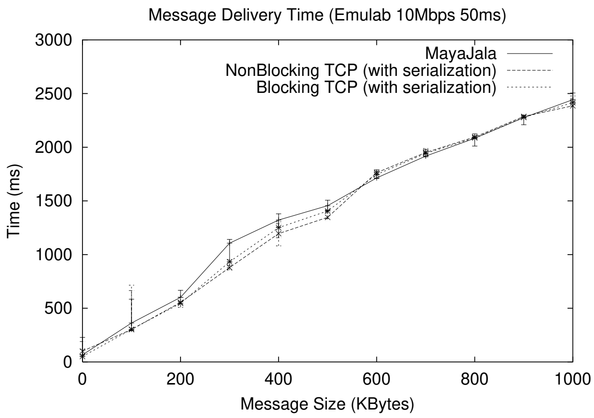

Figure 12: Message delivery time (Emulab).

8 Conclusions and Future Work

There are large number of communication paradigms that are useful for distributed applications and there is real value in identifying these communication paradigms precisely. We presented a model that describes the communication paradigms precisely as multiparty communication types. We also defined the type conformance for communication types, which allows the use of different conforming communication types in an application. Then we presented a middleware system, MayaJala, that facilitates the implementation, deployment, and use of communication types in distributed applications.

MayaJala simplifies both the application code and the agent code that implements a communication type. We compared two agents against similar overlay implementations in Macedon, which is considered as a system that facilitates concise implementations of overlays, and showed that MayaJala agents can be written with fewer lines of code. We also showed that the overhead of MayaJala is minimal compared to the Internet scale communication costs.

The design of MayaJala is very modular. The components can evolve independently. Most importantly the agents can be implemented independent of the applications that use them and the communication types are the contracts between the application developers and the implementors of the agents. In other words, the agents modularize the communication and increase the option value [4] of the application.

While the design of MayaJala is based on the black box model there is a disconnection between the communication types defined as logical formulae and the agents that implements them; the agents do not encode the communication types and nor do applications specify the communication types in the code. The communication type system is a tool available to the designers of the applications and there is no programming language representation of them—this is similar to design patterns, which do not have a programming language representation. It is doubtful whether encoding communication types can help automatic *type checking* of agents against the type specification in the applications since the communication types as specified in this work use first order predicate logic, which is undecidable. Even with such limitations we demonstrated the utility of communication types.

There is a body of work that uses distributed hash table (DHT) based overlay networks 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 [24]. 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 in the current version) that provides richer communication facilities.

The future research on MayaJala includes addressing security issues, using dynamic class loaders for agents, incorporating techniques in measuring link capacity and latency, and improving the communication type system.

The utility of MayaJala can be fully explored only if it is widely used by independent programmers. We plan to release MayaJala as an open source project in the near future and hope to keep a repository of implemented communication types.

References

[1] Y. Amir, C. Danilov, and J. R. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proc. of the 2000 International Conference on Dependable Systems and Networks*, pages 327–336. IEEE Computer Society, 2000.

[2] M. H. Ammar. Probabilistic multicast: Generalizing the multicast paradigm to improve scalability. In *Proc. of the Networking for Global Communciation. Volume 2*, pages 848–855, Los Alamitos, CA, USA, June 1994. IEEE Computer Society.

[3] B. R. Badrinath and P. Sudame. Gathercast: The Design and Implementation of a Programmable Aggregation Mechanism for the Internet. In *Proc. ICCCN*, pages 206–213. IEEE, Oct. 2000.

[4] C. Y. Baldwin and K. B. Clark. Modularity in the design of complex engineering systems. Working Paper Series 04-055, 2004, Harvard Business School, Jan. 2004.

[5] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of SIGCOMM '02*, pages 205–217. ACM Press, 2002.

[6] M. Bergman, J. Moor, and J. Nelson. *The Logic Book: Second Edition*. McGraw-Hill, 1990.

[7] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and F. Zongming. Application-layer anycasting. In *Proc. of INFOCOM 97*, pages 1388 –1396. IEEE, 1997.

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

[9] K. Birman, R. Constable, M. Hayden, J. Hickey, C. Kreitz, R. van Renesse, O. Rodeh, and W. Vogels. The horus and ensemble projects: Accomplishments and limitations. In *Proc. of DARPA Information Survivability Conference and Exposition*, pages 149–160, 2000.

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

[11] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distributed and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.

[12] K. L. Calvert, J. Griffioen, A. Sehgal, and S. Wen. Concast: Design and implementation of a new network service. In *Proc. of the Seventh Annual International Conference on Network Protocols*, pages 335–344. IEEE Computer Society, 1999.

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

[14] Y. Chae, E. W. Zegura, and H. Delalic. PAMcast: Programmable Any-Multicast for Scalable Message Delivery. In *Proc. of OPENARCH*, pages 25–36, June 2002.

[15] S. Y. Cheung and A. Kumar. Efficient quorumcast routing algorithms. In *Proc. of the Networking for Global Communciation. Volume 2*, pages 840–847, Los Alamitos, CA, USA, June 1994. IEEE Computer Society.

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

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

[18] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. *LNCS*, 1908:346–, 2000.

[19] N. Francez and B. Hailpern. Script: A communication abstraction mechanism. In *PODC*, pages 213–227, 1983.

[20] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. Promise: peer-to-peer media streaming using collectcast. In *Proc. of the eleventh ACM international conference on Multimedia*, pages 45–54. ACM Press, 2003.

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

[22] 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 *Proc. of OSDI 2000*, pages 197–212. USENIX, 2000.

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

[24] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proc. of IPTPS04*, San Diego, CA, February 2004.

[25] D. Katabi and J. Wroclawski. A framework for scalable global IP-anycast (GIA). In *Proc. of SIGCOMM '00*, pages 3–15. ACM Press, 2000.

[26] K. Lai and M. Baker. Measuring bandwidth. In *Proc. of INFOCOM*, pages 235–245, 1999.

[27] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In *Proc. of WARGC 2001*, pages 37–42. IEEE, 2001.

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

[29] 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.

[30] A. Nakao, L. Peterson, and A. 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.

[31] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. Internet Engineering Task Force: RFC 1546, November 1993.

[32] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proc. of USITS '01*, pages 49–60, San Francisco, CA, Mar. 2001. USENIX.

[33] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: a general-purpose programming language. *Softw. Pract. Exper.*, 21(1):91–118, 1991.

[34] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. of NSDI 2004*, pages 267–280. USENIX, 2004.

[35] J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference '03*, Miami, Florida, October 2003.

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