Multiparty Communication Types for Distributed Applications

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

Communication is an important and complex component of distributed applications. It requires considerable effort and expertise to implement communication patterns in distributed applications. Therefore, it is prudent to separate the two tasks of implementing the communication and implementing the application's main functionality which needs the communication. Traditionally this separation is achieved through a standard interface. A good example is the Message Passing Interface (MPI) for message-passing parallel programs. It takes considerable experience, effort and general agreement in the community to define such an interface. However, a standard interface is not flexible enough for the rapidly changing requirements of distributed applications. We propose that the separation of communication and application specific functionality should be achieved through the abstraction of communication types. In this paper we present a communication type system for multiparty communication. The type system can be used to express the communication requirements of an application, describe an implementation of a communication type, and make a match between these two. Our type system is only a single component of a framework for multiparty communication that we are developing.

1. Introduction

Many distributed applications exhibit interesting communication patterns that involve a set of nodes; for example multicast. We use the term *multiparty* communication to refer to this type of communication patterns. The Internet, which provides the communication medium for most of today's distributed applications, does not natively support such multiparty communication patterns in the network—multicast is an exception. Even multicast, which is part of the IP [10] protocol suite, is not universally available. This forces programmers to create their own solutions such as application layer multicast [16]. While this task is not impossible, it requires considerable effort and knowledge. Once implemented, such a communication pattern could be made available to other programs to prevent the redundant effort of implementing the same communication pattern again and again. Therefore a good design of distributed application should separate the concern of implementing the communication pattern from the concern of implementing the application specific functionality. The implementation of the communication pattern, which requires certain skills and knowledge, can be offloaded from the application programmer. Applications can simply be written assuming an implementation for the required communication pattern exists.

Traditionally such a separation between the application and the communication is achieved through a standard interface. The Message Passing Interface (MPI) [15] is a fine example. MPI defines a set of standard interfaces for several collective communication routines. Parallel programmers can take advantage of these interfaces and develop applications without worrying about the implementation. For example an application that uses `MPI_Bcast()` can be deployed over a cluster or a super computer simply by linking with a different implementation of MPI. The performance may change but the functionality remains the same.

Standard interfaces, while extremely useful, have a drawback—inflexibility. We cannot define interfaces as we want without the agreement from a larger community. New and arbitrary communication patterns cannot be easily accommodated into a standard. Distributed applications have diverse communication needs which should not be confined by a standard. Furthermore, the same implementation can be used with different interfaces—providing a different interface to an implementation of a communication pattern is not as complicated as the implementation of the communication pattern itself. Also, in the design phase of the application it is convenient not to be confined by an interface (or an implementation).

To address the limitations of a standard interface, we

propose that the separation between the application and the communication should be achieved through an abstraction of a communication type. We define a communication type as a mathematical object that names and defines the communication pattern precisely. It is independent of any interface and any implementation. In this paper we present a communication type system that can be used to express the requirements of an application, the functionality provided by an implementation, and to make a match between the requirements and the functionality.

This paper is organized as follows. In the next section we discuss several multiparty communication types and some systems that provide them. Note that for this discussion we use a loose definition of a communication type—i.e., a name that describes a communication pattern. In Section 3 we discuss the folly of using such everyday terms to describe the communication types and give a precise definition of communication types. Finally, in Section 4 we present our conclusions.

2. Multiparty Communication Types

In this section we show that a wide variety of multiparty communication types are in use or have been proposed.

The Message Passing Interface (MPI) [15] provides a variety of multiparty communication types—in MPI parlance collective communication primitives. As the name suggests, the MPI collective communication primitives are truly collective—all the nodes in the group (communicator [15]) 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. 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 [11]. This defeats the goal of many distributed applications, which are distributed to avoid a single point of failure.

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 [10], 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 [8], Overcast [12], ALMI [16] and Bayeux [19]. 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 [9, 13]. 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. [5] 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 [4] and Gathercast [2] are two implementations of this pattern as a network service in wide area networks. Calvert et al. [4] 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 Gathercast 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. [6] 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 [5]. Chae et al. [6] 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. [7] 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. [18] 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. The Type System

Loosely defined communication types could raise more questions than provide 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 [18] 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.

3.1. Definition and Examples

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., $m_0, m_1 \ldots m_n$). 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, $M = \langle M_0, \ldots, M_n \rangle$. 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_m as follows:

$$t_m(M, \overline{M}) \Leftrightarrow$$
$$\exists k[\ (M_k \neq \emptyset) \wedge \forall i[\ (\overline{M_i} = M_k) \wedge (i \neq k \Rightarrow M_i = \emptyset)]].$$

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 $t_m(M, \overline{M})$ is true.

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

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

t_{bm} 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:

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

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. [17] 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. [17] 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. Communication and Computation

We identify two important categories of communication types; the communication types that do not create new messages and the communication types that create new messages. We give the definition of these two categories as follows.

Let $\cup(M) = \bigcup_i M_i$ be the set of messages in the system. If $t(M,\overline{M}) \Rightarrow \cup(M) = \cup(\overline{M})$ then t is a *pure communication type*. That is, messages are neither created nor destroyed by the communication type t and t only distributes the messages among the nodes. Communication types like multicast, gather and scatter fall into this category.

On the other hand if $t(M,\overline{M}) \Rightarrow \cup(M) \subset \cup(\overline{M})$ then t is a communication type that *integrates computation with the communication*. That is, new messages are added to the system because of the communication, but still messages do not get destroyed. In the next section we discuss a communication type that belongs to this category.

3.3. Message Ordering

We discuss message ordering with reference to two scenarios. The first scenario is where a single source sends messages to one or more recipients. A case in point is multicast. Are messages delivered in the same order as they were sent? This question often arises because we tend to think in terms of some communication environment. For example, thinking about the messages in terms of packets. In our type system a message is indivisible and the question of the order does not arise, provided that we think in terms of the logical indivisible message units that we are interested in. For example, if we have a stream of data to send and we want the whole stream to be delivered in order then the message, as far as the type system is concerned, is the whole data stream. Also note that even though we used the terms like send and receive to describe this idea, the type system is void of such implementation details.

In the second scenario there are multiple senders and multiple receivers and we want the guarantee that all the nodes received the messages in the same order—a total order in message delivery. To explain how the type system describes such a communication type we take the *guaranteed delivery all-to-all exchange* message type, t_{aa} as an example. We define t_{aa} as follows.

$$t_{aa}(M,\overline{M}) \Leftrightarrow \forall i\ [\overline{M_i} = M_i \cup \{P(M)\}].$$

After the communication each node has its original message set and a new message $P(M)$. $P(M)$ is a permutation of messages in the sets $M_0,\ldots,M_{|M|-1}$. Note that all the nodes have the same message $P(M)$ after the communication. This guarantees total order message delivery in all the nodes.

Note that there is a new message, $P(M)$, in the system after the communication. This should be the case because by receiving $P(M)$ a node not only gets all the messages that were in the system before, it also gets a new message which is the knowledge that all the other nodes received the messages in the same order. This communication type belongs to the category of the communication types that integrates communication with the computation. A similar formula can be used to describe communication types such as reduction.

3.4. 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.

3.5. Sub Types

Given two communication types t_a and t_b, if $t_a(M,\overline{M}) \rightarrow t_b(M,\overline{M})$ then t_a is a sub type of t_b (and t_b is a super type of t_a). For example, t_m is a sub type of t_{bm}. We can prove that $t_m(M,\overline{M}) \rightarrow t_{bm}(M,\overline{M})$. A consequence of this is that an implementation of t_m can be used whenever an implementation of t_{bm} is required. In general, whenever an application needs a communication type it can use an implementation of a sub type of the required type.

Interestingly, t_{any} is also a subtype of t_{bm} since we can prove that $t_{any}(M,\overline{M}) \rightarrow t_{bm}(M,\overline{M})$. This may not match our intuitive notion of the relationship between best-effort multicast and Anycast, but shows one of the values of the type system: our intuitive notions of the meaning of a communication pattern are often incorrect.

4. Conclusions

Considerable effort and expertise are required to implement a multiparty communication pattern over a network such as the Internet. We argued that it is prudent to separate the implementation of the communication from the implementation of the application specific functionality. Traditionally such a separation is achieved through a standard interface. We argued that a standard interface is not flexible enough to accommodate rapidly changing and wide variety of communication requirements of distributed applications. We argued that the separation of the communication and the application specific functionality should be achieved through the abstraction of a communication type.

We presented a communication type system that provides the connection between the requirements of the application and the functionality provided by a module that implements the communication. The type system is independent of any notion of an implementation or an interface.

The type system is still evolving and it is a part of a framework that we are developing to support multiparty communication in distributed applications. Apart from the type system the framework also provides a programming model and a middleware system to support the implementation, the deployment and the use of multiparty communication. Detailed discussion of the framework can be found in [14].

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 (ICCCN)*, 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] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

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

[14] C. Keppitiyagama and N. C. Hutchinson. Mayajala: A framework for multiparty communication. Technical Report TR-2003-10, Department of Computer Science, University of British Columbia, Vancouver, Canada, Sept. 2003.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.