Group Communication*

Idit Keidar
MIT Laboratory for Computer Science

June 2000, revised April 2001

Traditional communication models and protocols like TCP (see Transmission Control Protocol - Internet Protocol (TCP-IP)) support point to point communication. Such protocols were designed for applications involving communication between no more than two processes at a time, usually a client and a server. Many modern applications do not adhere to this communication model. Consider for example an on-line game being played by several participants around the world; or a multi-media conference in which users see each other, talk to each other and also write on a shared white board. These applications involve more than two users exchanging information. They require multi-point to multi-point communication.

Group communication is a means for providing multi-point to multi-point communication, by organizing processes in groups [1, 2, 3, 4]. A *process* is an instance of an executing program at a certain location. A *group* (or *process group*) is a set of processes which are *members* of the group. So, for example, a group can consist of the users playing an on-line game with each other, in the same virtual universe. Another group can consist of the participants in a multi-media conference. Each group is associated with a logical name (or address). Processes communicate with group members by sending a message targeted to the group name; the group communication service delivers the message to the group members. Sending a message to multiple recipients in this way is called *multicast* (see Multicast Communication Systems).

Groups are usually dynamic, in the sense that the set of group members continuously changes. Processes may choose when they wish to join or leave a group. For example, users can independently start or stop playing a game at any time.

Typical group communication services

In the early 1980s, many researchers began to develop replicated databases and file systems which required the coordination of multiple copies of the data. These replicated systems used mechanisms for multi-point to multi-point communication among groups of processes internally. However, such mechanisms were not exported as separate group communication services. Group communication services per se began to emerge in the late 1980s. Since then, a large variety of group communication services have been developed.

Group multicast services range from *best-effort* unreliable multicast (see Best-effort) to *reliable multicast*, which ensures that messages sent among non-faulty processes are not lost; from *unordered* services that deliver messages in arbitrarily different orders to different group members, to *totally ordered* services that deliver messages in the same order to all the group members. For example, if

*Chapter in the *Encyclopedia of Distributed Computing*, Joseph Urban and Partha Dasgupta, editors, Kluwer Academic Publishers. To be published.

processes p and q both receive multicasts m and n, then with a totally ordered service, both receive m before n, or n before m. A reliable totally ordered service that delivers all the messages in the same order to all the group members is called *Atomic Broadcast* (see Atomic Broadcast).

Most group communication services feature a *group membership* service which tracks the list of the processes belonging to the group as it evolves over time, and reports these changes to the group members. Group communication systems that provide membership services often support semantics called Virtual Synchrony (see Virtual Synchrony), which help processes remain synchronized when failures occur. More details are presented in the section on group membership.

Historically, the evolution of group communication systems began at extreme ends of the spectrum. On the unreliable end, the basis for supporting multicast on the Internet was laid by Deering around 1989 with the introduction of the IP multicast extensions [5]. (See Multicast Communication Systems). On the reliable end, the Isis Reliable Computing Toolkit project was developed in 1987 at Cornell University [2]. Isis provided group membership and Virtual Synchrony semantics, as well as several reliable multicast services with different ordering guarantees, for example, atomic broadcast.

Unreliable group multicast systems typically provide good scalability, up to tens of thousands of nodes, but their semantics are generally too weak for application developers to depend upon. Messages are subject to long and unpredictable transmission delays, message loss, and out of order delivery. Processes may crash and network links may fail. Such failures are hard to detect when the communication delays are unpredictable and messages can be lost. These problems make it extremely difficult to build network applications that behave predictably when failures occur.

Consider the on-line game example, a group of players can use group communication to inform each other of their actions, as illustrated in Figure 1. Assume that some player, Alice, shoots at another player, Bob. If the information about the shooting is sent in a message, what would happen if the message did not reach Bob at all? To avoid this, we could use a reliable multicast service that ensures that a message from a correct process reach all the correct participants. But then what would happen if Alice failed shortly after shooting at Bob, and Bob did not receive Alice's last message, while another player, Carol, did? And assume Alice wishes to re-join the game (by running the program again after its failure). How would Alice learn about the others' current locations when she re-joined? Or even in absence of failures, what would happen if Bob thought that he has moved away before the shot but Alice thought that he attempted to move only after the shot? These problems illustrate the difficulty of building reliable distributed applications, and the need for a more powerful form of group communication.

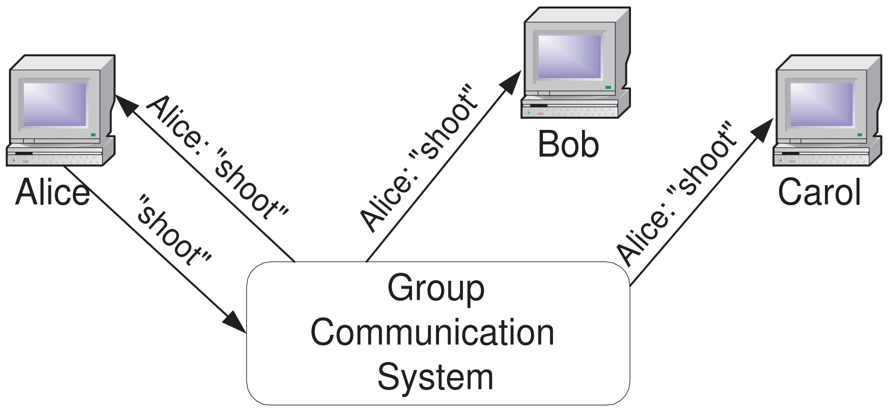

Figure 1: Using group communication in an on-line game.

Reliable group communication systems that support Virtual Synchrony semantics greatly facilitate the development of fault-tolerant distributed applications. These systems serve as building blocks, or "middleware" for reliable distributed applications. They abstract away some of the difficulties of the network, such as those illustrated above. They create the illusion (or abstraction) of an idealized network, where messages are never lost and never arrive out-of-order; where there is a simple way to detect failures; and where failures are observed as happening "at the same time" by all the non-faulty participants. This abstraction is especially useful for applications that maintain replicated state of some sort (see the section on state machine replication), like the same picture of a game in the example above. The main drawback of group communication systems that provide such semantics is that they do not scale well beyond a few hundreds of processes.

In recent years, we have been witnessing increasing convergence of the two approaches. It is clear that both scalability and reliability are important considerations, and current research focuses on balancing the two. Scalable reliable multicast services for the Internet environment have been developed, and improving such services is a very active research area (see Multicast Communication Systems). At the same time, research on group communication systems that support Virtual Synchrony focuses on scalability, on deployment in the Internet environment, and also on provision of a wide range of multicast services with different semantics, including services that preserve the *quality of service (QoS)* (see Quality of Service) of the underlying network.

Group membership

The task of a *group membership service* is to maintain a list of the currently active and connected processes in the group. When this list changes (with new members joining and old ones departing or failing), the group membership service reports the change to the group members. The output of the membership service is called a *view*, consisting of the list of the currently active and connected members in the group and a unique identifier. The membership service strives to deliver the same view (consisting of the same member list and the same identifier) to mutually connected members.

When communication links fail, the network may partition into multiple mutually disconnected *network components*. Different group membership services handle such situations in different ways. *Primary component* membership services allow only one network component, called the primary component, to continue running the service, whereas processes in other network components are considered faulty. A *partitionable* membership service allows processes in all components to continue running the service. With primary component membership, each non-faulty member observes the same sequence of membership views, starting with the membership of the group at the time it joined, and continuing until it leaves the group, crashes, or disconnects from the primary component. In contrast, partitionable membership services allow multiple disjoint views of the same group to exist concurrently in different network components.

The Isis group communication service supports primary component membership. Partitionable membership was first introduced as part of the Hebrew University of Jerusalem's Transis project (see Transis), and was later supported by other group communication systems like the University of California Santa Barbara's Totem project (see The Totem System), The University of Bologna's Relacs project, as well as by Isis' successors at Cornell: Horus and Ensemble. See related papers in [1]; for a survey, see [4].

Group membership lies at the core of the Virtual Synchrony execution model (see Virtual Synchrony). A key property of this model is "virtually synchronous delivery" [3, 4] (see Virtually Synchronous Delivery); this property specifies that if the two views are delivered consecutively to several processes, then exactly the same multicast messages are delivered to these processes between these two views. Assume for example that the group communication service delivers to Bob the view $\langle\{\text{Alice, Bob, Carol}\}, 1\rangle$ (recall that a view consists of a list of members and an identifier), then a multicast message "Alice shoots", and then the view $\langle\{\text{Bob, Carol}\}, 2\rangle$. Assume further that the group communication service also delivers to Carol the view $\langle\{\text{Alice, Bob, Carol}\}, 1\rangle$ followed by the view $\langle\{\text{Bob, Carol}\}, 2\rangle$. Then, the service must also deliver to Carol the message "Alice shoots" between these two views.

The Virtual Synchrony (see Virtual Synchrony) execution model was first presented in the context of a primary component membership service (in Isis). Several variations on Virtual Synchrony semantics for both primary component and partitionable membership have been suggested. Brevity precludes detailed discussion of the various semantics; for a comprehensive survey see [4].

State machine replication

One of the original motivations for group communication systems like Isis and for the Virtual Synchrony model was supporting replication using the *state machine* approach. This motivation encompasses a vast class of applications. In fact, replication occurs in most distributed systems.

Distributed systems typically maintain shared state of some sort. In some cases, the shared state can be large, like a database or a file system. In other cases, the shared state can be small – consisting, for example, of only the list of participants in a video conference, or the current locations of players in an on-line game. Shared states are usually *replicated* among a group of processes. Distributed systems employ replicated services to achieve fault-tolerance and to improve performance by placing replicas close to where service is needed.

When the replicated state is being updated, all the replicas must be modified in a consistent manner. To this end, a *replication management* protocol is employed. *State machine replication* [6, 7] (or *active replication*) is a common paradigm for replication management that has no centralized control. This paradigm models services as *deterministic state machines.*

Deterministic state machines provide a general model for defining services and their semantics. In this model, a service is defined as a *state machine* consisting of *state variables*, which encode its state, and *actions*, (or *commands*), which modify its state and/or produce output. Each action is implemented by a deterministic program which is executed atomically with respect to other actions. The state machine specifies the computation semantics of the service: outputs of a state machine are completely determined by the sequence of actions it executes.

With state machine replication, replicas are represented as identical deterministic state machines. The main idea behind state machine replication is as follows: if all replicas execute the same sequence of actions, then all replicas remain in a consistent state, and produce identical sequences of outputs. In the presence of faults, the above requirement is imposed on all non-faulty replicas.

Atomic broadcast along with Virtual Synchrony provide a convenient framework for state machine replication: replicas are organized as a group, and actions are invoked in response to the delivery of multicast messages and views. Since messages and views are delivered in the same order to all non-faulty replicas, consistency is preserved.

Let us re-visit the on-line game example above. If Alice shoots at Bob, and at the same time, Bob moves away, then atomic broadcast ensures that Alice's shot and Bob's movement away from the shot are reported to all the participants in the same order. Assume the movement is reported first, then for every player, the state machine implementing the game executes the "move" action before executing the "shoot" action. If the machine has a state variable representing the fact that Bob is alive, then this variable remains set to true for all instances of the state machine.

Using group membership, Bob and Carol observe Alice's failure by delivering a view that excludes her. The key property of Virtual Synchrony guarantees that either both Bob and Carol learn of Alice's shot before detecting her failure, or else neither one of them does. Group communication systems often provide a *state transfer* service (see State Transfer), which facilitates the process of bringing Alice up-to-date when she re-joins.

This article illustrates the power of group communication, and its utility for building distributed applications. Brevity precludes discussion of the full spectrum of group communication services, applications, and implementations. More information can be found in [1, 3, 4].

References

1. Communications of the ACM 39(4), special issue on Group Communications Systems, April 1996.

2. K. Birman, The Process Group Approach to Reliable Distributed Computing, Communications of the ACM, 36 (12), 37-53, 1993.

3. K. Birman, Building Secure and Reliable Network Applications, Manning Publishing and Prentice Hall, 1997.

4. G. V. Chockler, I. Keidar, and R. Vitenberg, Group Communication Specifications: A Comprehensive Study, ACM Computing Surveys, to appear.

5. S. Deering, Host Extensions for IP Multicasting, RFC 1112, August 1989.

6. L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 21(7), 558-565, July 1978.

7. F. B. Schneider, Implementing Fault Tolerant Services Using The State Machine Approach: A Tutorial, ACM Computing Surveys 22(4), 299-319, December, 1990.