Lectures on distributed systems

Group Communication

Paul Krzyzanowski

Introduction

Remote procedure calls assume the existence of two parties: a client and a server. This, as well as the socket-based communication we looked at earlier, is an example of point-to-point, or unicast, communication. Sometimes, however, we want one-to-many, or **group**, communication.

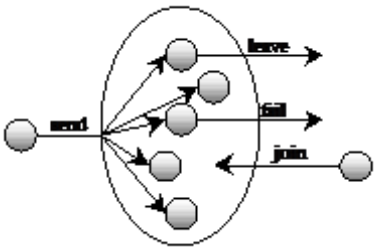

Figure 1. Group dynamics

Groups are dynamic (Figure 1). They may be created and destroyed. Processes may join or leave groups and processes may belong to multiple groups. An analogy to group communication is the concept of a mailing list. A sender sends a message to one party (the mailing list) and multiple users (members of the list) receive the message. *Groups allow processes to deal with collections of processes as one abstraction.* Ideally, a process should only send a message to a group and need not know or care who its members are.

Implementing group communication

Group communication can be implemented in several ways. Hardware support for multicasting allows the software to request the hardware to join a multicast group. Messages sent to the multicast address will be received by all network cards listening on that group(s) (Figure 2). If the hardware does not support multicasting, an alternative is to use hardware broadcast and software filters at the receivers. Each message is tagged with a multicast address. The software processing the incoming messages extracts this address and compares it with its list of multicast addresses that it should accept. If it is not on the list, the message is simply dropped (Figure 3). While this method generates overhead for machines that are not members of the group, it requires the sender to only send out a single message.

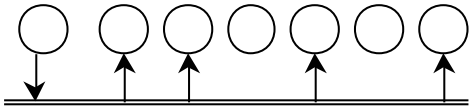

Figure 2. Multicast

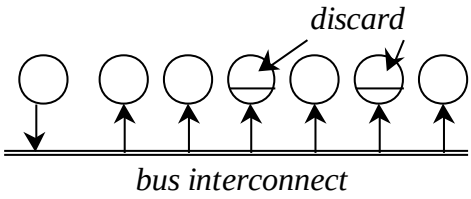

Figure 3. Simulated multicast via broadcast

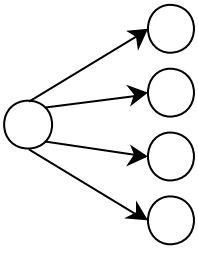

Figure 4. Simulating group communication with multiple unicasts

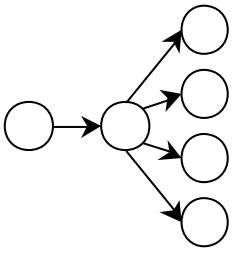

Figure 5. Simulating group communication with a central coordinator

A final implementation option is to simulate multicasting completely in software. In this case, a separate message will be sent to each receiver. This can be implemented in two ways. The sending machine can know all the members of the group and send the same message to each group member (Figure 4). Alternatively, some machine can be designated as a group coordinator: a central point for group membership information (Figure 5). The sender will send one message to the group coordinator, which then iterates over each group member and sends the message to each member.

As with unicast communication, group communication also requires a transport-level protocol. Even with hardware support, there must be a mechanism for directing data to the interested process(es)

Design issues

A number of design alternatives for group communication are available. These will affect how the groups behave and send messages.

Closed group vs. open group	With **closed groups**, only the group members may send a message to the group. This is useful when multiple processes need to communicate with others in solving a problem, such as parallel processing applications. The alternative is **open groups**, where non-members can send a message to a group. An example use of this type of group is an implementation of a replicated server (such as a redundant file system).
Peer groups vs. hierarchical groups	With **peer groups**, every member communicates with each other. The benefits are that this is a decentralized, symmetric system with no point of failure. However, decision making may be complex since all decisions must be made collectively (a vote may have to be taken). The alternative is **hierarchical groups**, in which one member plays the role of a group coordinator. The coordinator makes decisions on who carries out requests. Decision making is simplified since it is centralized. The downside is that this is a centralized, asymmetric system and therefore has a single point of failure.
centralized group membership vs. distributed membership	If control of group membership is **centralized**, we will have one **group server** that is responsible for getting all membership requests. It maintains a database of group members. This is easy to implement but suffers from the problem that centralized systems share – a single point of failure. The alternative mechanism is to manage group membership in a distributed way where all group members receive messages announcing new members (or the leaving of members).

Several problems can arise in managing group membership. Suppose a group member crashes. It effectively leaves the group without sending any form of message informing others that it left the group. Other members must somehow discover that it is missing.

Leaving and joining a group must be synchronous with message delivery. No messages should be received by a member after leaving a group. This is easier to achieve if a group coordinator/group server is used for message delivery and membership management.

A final design issue is that if the machines and/or the network die so the group cannot function, how are things restarted?

Send/receive primitives

Remote procedure calls (RPC) were, for many applications, more convenient and intuitive than the send/receive (write/read) model provided by sockets. Remote procedure calls, however, do not lend themselves to group communication. RPC is based on a function call model wherein a procedure is called and a value returned as a result. If we try to apply this to group communication, one message is sent to the group to invoke the procedure. The return value is not clear now, since *every* member of the group may generate one. RPC just does not expect this behavior. We have to fall back on send/receive primitives when working with a group.

Atomic multicast

One desirable property for certain types of group communication is that of ensuring that *all* group members get a message. More specifically, if a message is sent to a group and one member receives it, that member can be sure that *all* members will get the message. This is an *all or nothing* property: it either arrives correctly at all members or else *no* member receives the message. There will never be a situation where some members receive the message and others do not. This property is known as **atomicity** and this type of multicast is called an **atomic multicast.** An atomic multicast is appealing because it makes application design easier in that there is one less thing to worry about – missing or partially delivered messages.

While this property is desirable, it is not easy to achieve. The only way to be sure that a destination received a message is to have it send back an acknowledgement message upon message receipt. This is prone to problems since some replies can be lost, the sender may have crashed after sending the message and cannot process the replies, or the receiver crashed before it could send a reply. What we need to do to achieve an atomic multicast is to ensure that we can deliver messages even with process failures.

There are several ways that we can achieve this. One way is to use the concept of a persistent log from database systems. The persistent log is simply a series of messages written onto a disk or some non-volatile memory so that it could be recovered even if the process dies. Should a process die, it is responsible for reading the log when it comes up again.

In this system, the sender sends messages to all members of the group and waits for an acknowledgement from each member. The sender saves a copy of the message in the log and also logs each acknowledgement it has received. This way, even if it dies, it can resume where it left off once the process is restarted. If an acknowledgement has not been received from a member, the sender will retransmit periodically until the member acknowledges the message. On the receiving side, a group member logs the received message into its persistent log upon

receiving the message and prior to sending the acknowledgement. Even if the member dies now, it will have the message when it restarts. When all members have acknowledged receipt of the message, the sender can then send a "deliver" message, instructing each member to deliver the message to the higher layers of the software that will process the message.

This solution is somewhat troublesome to implement in terms of logging and recovering from failed processes. The essential point is that the protocol must account for the sender crashing after it sent some or all of the messages and for receivers that may be dead at any point during the multicast.

Reliable multicast

A compromise to atomic multicast is to assume that the sending machine will remain alive to ensure that a message was sent out to all members of the group. This is called a **reliable multicast**. It is a best-effort attempt at reliability but makes no guarantees in the case where the sender is unable to transmit or receive messages to other group members. One implementation can be:

1. Set a long timer, T_L. This will be used to detect an unresponding machine.
2. Set a shorter timer, T_S. This will be used to detect lost messages or lost acknowledgements.
3. Send a message to each group member.
4. Wait for an acknowledgement message from each group member.
5. If timer T_S goes off, then retransmit the message to members that have not responded, reset the timer, and wait.
6. If timer T_L goes off, then label the unresponding machines as "failed" and remove them from the group.

In the best case, if multicast or broadcast facilities are available, the sender needs to only send one message. If these facilities are not available, they can be simulated:

```
for ( dest in group)
    send(dest, msg)
```

Each recipient sends one message as an acknowledgement.

We can try to increase performance by decreasing the number of messages sent. The sender maintains a count of the number of messages sent. This count is appended to each message sent and acts as a message sequence number. Recipients send no acknowledgement message unless the sequence number indicates that a message was missed. This is known as a **negative acknowledgement** protocol. The sender is responsible for keeping copies of old messages for retransmission. The problem with this protocol is that the sender has no way to detect that a machine is no longer responding.

Unreliable multicast

If the reliable multicast is deemed too costly, the next step down is the **unreliable multicast**. This is the basic multicast in which a message is sent and the process just hopes that it arrives at all destinations. It is useful for services that don't require reliability (e.g. multicast video and audio). It is also useful in cases when the sender does not know the identity of the group members.

If multicast or broadcast facilities are available, the sender needs to only send one message. The recipients need to send nothing. If these facilities are not available, they can be simulated as mentioned above.

Message ordering

To make group communication easy to use and understand, two properties are desirable:

- atomicity: message arrives everywhere
- first-in-first-out (FIFO) message ordering: consistent message ordering.

Suppose there is a group of four machines {0, 1, 2, 3}. Machines 0 and 1 send messages simultaneously via multiple unicasts:

$0 \rightarrow 1$, $0 \rightarrow 2$, $0 \rightarrow 3$

$1 \rightarrow 0$, $1 \rightarrow 2$, $1 \rightarrow 3$

If the messages appear on the network in the following chronological sequence:

$\{0 \rightarrow 1\}$, $\{1 \rightarrow 0\}$, $\{1 \rightarrow 2\}$, $\{0 \rightarrow 2\}$, $\{0 \rightarrow 3\}$, $\{1 \rightarrow 3\}$

then machine 2 receives a message from machine 1 first, followed by a message from machine 0. Machine 3, however, receives a message from machine 0 first, followed by a message from machine 1. If machines 0 and 1 were trying to update the same record in a database, 2 and 3 could end up with different values. To avoid confusion and potential problems, it is desirable to have all messages arrive in the exact orders sent. This is known as **global time ordering**. It is not always easy to implement global time ordering. A compromise it to say that if two messages are close together, the system picks one of them as being "first." All messages arrive at all group members in the same order (which may or may not be the exact order sent). This compromise is called **consistent time ordering** or **total ordering**.

One algorithm for achieving total ordering is:

1. Assign a unique totally sequenced message ID[1] to each message.
2. Each message is regarded as stable at an element if no message with a lower ID is expected to arrive. When messages can arrive out of order, the system will accept such messages but not forward them to the application. A message is stable at an element when the system has received all earlier messages and passed them on to the receiving process. Any message that is stable at an element can be immediately passed on to the receiving process. This ensures in-order delivery. Any other messages are buffered until the out-of-order messages are received.
3. The communications driver passes only stable at an element messages to the application, passing the message with he lowest ID first.
4. Each member saves all messages in a queue for delivery to applications.

One problem that arises in implementing this protocol is that of generating a message identifier since we need a *shared* sequence of identifiers. A few solutions can be adopted:

[1] By a *totally sequenced message ID* we mean that all members of the group get unique, chronologically increasing sequence numbers.

- Use a **sequencer**, a common process to which all multicast messages are sent. The sequencer receives a message, attaches a sequence number, and then resends the message to the group members.

- Use a sequence number server. A process will first contact the sequence number server to request a sequence number. The process will then attach the sequence number to the message and multicast it.

- Alternatively, one can come up with a distributed protocol for generating unique, monotonically increasing message identifiers.

We can relax ordering rules further and dictate that ordering will be preserved only amongst *related* messages. Unrelated messages may be received in a different order on different systems. This *partial ordering* is known as **causal ordering** and the concept of *related messages* refers to Lamport message ordering and its *happened-before* relationship. Here, concurrent messages will not be ordered. Messages are sequenced strictly by their Lamport timestamps.

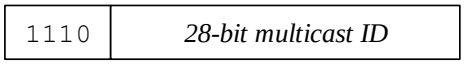

Figure 5. A class D IP address

Relaxing the rules even more, we can decide that ordering does not matter at all and messages can be received in a different order at different machines. However, we can provide a special message type: a *synchronization* (*sync*) primitive that can be sent to ensure that any pending messages are processed before any additional (post-sync) messages will be accepted. This means that if a message is sent, it will be processed by all members before the synchronization operation. Any message sent after a member sends a sync message will be processed by all members after the sync. Message delivery is not split on either side of the sync. A sync is also known as **barrier**. This type of message ordering is known as **sync ordering**.

Finally, the most relaxed form of message delivery is the **unordered multicast**. Messages can be delivered in a different order to different members. We may impose **sequential ordering per source**, which means that all messages sent from one member will be received in the order sent by all members, although members may receive different interleaved messages from others.

IP multicasting

As an example of a commonly-used multicasting protocol, we can consider IP multicasting. Multicasting under the Internet Protocol is performed by addressing IP packets with a *multicast address*. The class D network was created for this. A class D address contains four leading bits of `1110` followed by a 28-bit multicast ID number. This spans the IP addresses from 224.0.0.0 through 239.255.255.255 (Figure 5). The set of all machines listening to a particular multicast address make up a **host group**. These machines can span multiple physical networks. Membership is dynamic – a machine can leave or join a group at any time and there is no restriction on the number of hosts in a group. A machine does not have to be a member of the group to send messages to the group.

A multicast address may be chosen arbitrarily, but some well-known host group addresses are assigned by the IANA (Internet Assigned Numbers Authority). IANA information can be found in RFC 1340. This is similar to port numbers: arbitrary ports may be chosen but certain numbers are

reserved for known applications. For example, some well-known ports are 21 for FTP, 25 for SMTP, 80 for HTTP. Some well-known multicast addresses are 244.0.0.1 for all systems on this subnet, 224.0.1.2 for SGI's Dogfight, and 224.0.1.7 for the *Audionews* service.

LAN multicasting

Since IP is a logical network built on top of physical networks, it is worthwhile to examine how multicasting works on LAN cards (e.g. an ethernet card). LAN cards that support multicast support it in one of two ways:

1. Packets are filtered based on a hash value of the multicast hardware address (some unwanted packets may pass through because of hash collisions.

2. The LAN card supports a small, fixed number of multicast addresses on which to listen. If the host needs to receive more, the LAN card is put in a *multicast promiscuous* mode to receive *all* hardware multicast packets.

In either case, the device driver must check that the received packet is really the one that is needed. Even if the LAN card performed perfect filtering, there may still need to be a need to translate a 28-bit IP multicast ID to the hardware address (e.g. a 48-bit ethernet address). The translation of IP multicast ID numbers to ethernet addresses is defined by the IANA (Internet Assigned Numbers Authority), which decrees that the least significant 23 bits of the IP address are copied into an ethernet MAC address of the form 01:00:5e:xx:xx:xx.

IP multicasting on a single network

On a single physical network, the sender specifies a destination IP address that is a multicast address (class D). The device driver then converts this address to a corresponding ethernet address and uses this address in its hardware header (which envelopes the IP header). Now it sends out this multicast ethernet packet which contains a multicast IP packet within it.

When a process wishes to receive multicast packets, it notifies the IP layer that it wants to receive datagrams destined for a certain IP address. The device driver has to enable reception of ethernet packets that contain that IP multicast address. This action is known as **joining a multicast group**.

Upon receiving such packets, the device driver sends the IP packet to the IP layer, which must deliver a copy of the packet to all processes that belong to the group.

IP multicasting beyond the physical network

When IP packets flow through multiple physical networks, they go through routers which bridge one network to another. In the case of multicasting, a multicast-aware routed needs to know whether there are *any* hosts on a physical network that belong to a multicast group.

The *Internet Group Management Protocol* (IGMP, RFC 1112) is designed to accomplish this task. It is a simple datagram based protocol that is similar in principle to ICMP. Packets are fixed-size messages containing a 20-byte IP header, and 8 bytes of IGMP data. This data includes:

- 4-bit version number

- 4-bit operation type (1=query sent by router, 2=response)

- 16-bit checksum
- 32-bit IP class D address

The IGMP protocol works as follows:

- When the first process on a machine joins a multicast group, the machine sends an IGMP report stating that it is interested in that particular multicast address.
- Each multicast router broadcasts IGMP queries at regular intervals to see whether any machines still have processes belonging to any groups. One query is sent per network interface.
- When a machine receives an IGMP query, it sends one IGMP response packet for each group for which it is still interested in receiving packets.

The machine never sends a report when a process leaves the group (even if it is the last process that joined the group. Eventually the multicast router will stop forwarding packets to the network when it receives no IGMP responses for a particular multicast address.

References

Distributed Systems: Concepts and Design, G. Coulouris, J. Dollimore, T. Kindberg, Third Edition ©2001 Pearson Education Limited.

Distributed Systems: Concepts and Design, G. Coulouris, J. Dollimore, T. Kindberg, ©1996 Addison Wesley Longman, Ltd.

Distributed Systems, Principles & Paradigms, Andrew S. Tanenbaum and Maarten Van Steen. Second Edition ©2007 Pearson Education Limited.

Distributed Operating Systems, Andrew Tanenbaum, © 1995 Prentice Hall.

Modern Operating Systems, Andrew Tanenbaum, ©1992 Prentice Hall.

TCP/IP Illustrated – Volume 1 – the protocols, Richard Stevens, ©1994 Addison-Wesley.