Group Communication

A **group** is an operating system abstraction for a collective of related processes. A set of cooperative processes may, for example, form a group to provide an extendable, efficient, available and reliable service. The group abstraction allows member processes to perform computation on different hosts while providing support for communication and synchronisation between them.

The term **multicast** means the use of a single communication primitive to send a message to a specific set of processes rather than using a collection of individual point to point message primitives. This is in contrast with the term **broadcast** which means the message is addressed to every host or process.

A **consensus protocol** allows a group of participating processes to reach a common decision, based on their initial inputs, despite failures.

A **reliable multicast protocol** allows a group of processes to agree on a set of messages received by the group. Each message should be received by all members of the group or by none. The order of these messages may be important for some applications. A reliable multicast protocol is not concerned with message ordering, only message delivery guarantees. **Ordered delivery protocols** can be implemented on top of a reliable multicast service.

Multicast algorithms can be built on top of lower-level communication primitives such as point-to-point *sends* and *receives* or perhaps by availing of specific network mechanisms designed for this purpose.

The management of a group needs an efficient and reliable multicast communication mechanism to allow clients obtain services from the group and ensure consistency among servers in the presence of failures. Consider the following two scenarios:-

> A client wishes to obtain a service which can be performed by any member of the group without affecting the state of the service.
>
> A client wishes to obtain a service which must be performed by each member of the group.

In the first case, the client can accept a response to its multicast from any member of the group as long as at least one responds. The communication system need only guarantee delivery of the multicast to a nonfaulty process of the group on a best-effort basis. In the second case, the all-or-none atomic delivery requirements requires that the multicast needs to be buffered until it is committed and subsequently delivered to the application process, and so incurs additional latency.

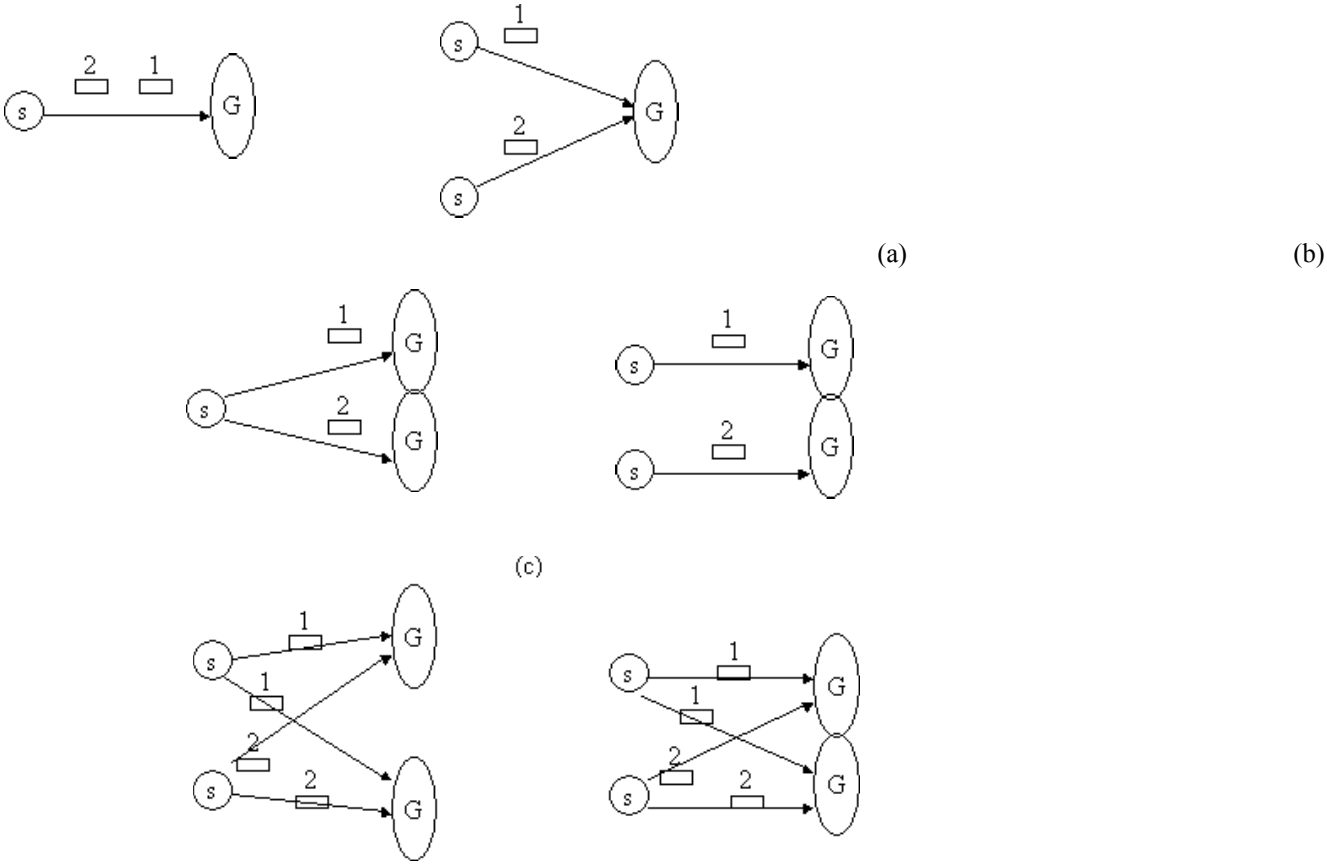
Failure may occur during a multicast at the recipient processes, the communication links or the originating process.

Failures at the recipient processes and on the communication links can be detected by the originating process using standard time-out mechanisms or message acknowledgements. The multicast can be aborted by the originator, or the service group membership may be dynamically adjusted to exclude the failed processes and the multicast can be continued.
If the originator fails during the multicast, there are two possible outcomes. Either the message has not have arrived at any destination or it has arrived at some. In the first case, no process can be aware of the originator's intention and so the multicast must be aborted. In the second case it may be possible to complete the multicast by selecting one of the recipients as the new originator. The recipients would have to buffer messages until safe for delivery in case they were called on for this role.

A reliable multicast protocol imposes no restriction on the order in which messages are delivered to group processes. Given that multicasts may be in progress by a number of originators simultaneously, the messages may arrive at different processes in a group in different orders. Also, a single originator may have a number of simultaneous multicasts in progress or may have issued a sequence of multicast messages whose ordering we might like preserved at the recipients. Ideally, multicast messages should be delivered instantaneously in the real-time order they were sent, but this is unrealistic as there is no global time and message transmission has a possibly significant and variable latency.

A number of possible scenarios are given below which may require different levels of ordering semantics. G and s represent groups and message sources. s may be inside or outside a group. Note that group membership may overlap with

other groups, that is, processes may be members of more than one group.



Ordered Reliable Multicasts

A **FIFO ordered** protocol guarantees that messages by the same sender are delivered in the order that they were sent. That is, *if a process multicasts a message m before it multicasts a message m', then no correct process receives m' unless it has previously received m*. To implement this, messages can be assigned sequence numbers which define an ordering on messages from a single source. Some applications may require the context of previously multicast messages from an originator before interpreting the originator's latest message correctly.

However, the content of message m may also depend on messages that the sender of m received from other sources before sending m. The application may require that the context which could have caused or affected the content of m be delivered at all destinations of m, before m. For example, in a network news application, user A broadcasts an article. User B at a different site receives the article and broadcasts a response. User C can only interpret the response if the original article is delivered first at their site. Two messages are said to be **causally related** if one message is generated after receipt of the other. **Causal order** is a strengthening of FIFO ordering which ensures that a message is not delivered until all messages it depends on have been delivered.

This causal dependence relation is more formally specified as follows:- An execution of a *multicast* or *receive* primitive by a process is called an *event*.

Event e causally precedes event f (i.e. *happened before*), (e®f), if an only if:

1. A process executes both e and f in that order, or
2. e is the multicast of message m and f is the receipt of m, or
3. there is an event h, such that e®h and h®f.

A causal protocol then guarantees that *if the broadcast of message m causally precedes the broadcast of m', then no correct process receives m' unless it has previously received m*.

The definition of causal ordering does not determine the delivery order of messages which are not causally related. Consider a replicated database application with two copies of a bank account x residing at different sites. A client side process at one site sends a multicast to the database to lodge £100 to account x. At another site simultaneously, a client side process initiates a multicast to add 10% interest to the current balance of x. For consistency, all database replicas should apply the two updates in the same sequence. As these two messages are not causally related, a causal broadcast would allow the update messages to x to be delivered in different sequences at the replicas.

Total Ordering guarantees that all correct processes receive all messages in the same order. That is, *if correct processes p and q both receive messages m and m', then p receives m before m' if and only if q receives m before m'*. The multicast is atomic across all members of the group.

Note that this definition of a totally ordered broadcast does not require that messages be delivered in Causal Order or even FIFO Order, so it is not stronger than these orderings.
For example, if a process suffers a transient failure during the broadcast of message m, and subsequently broadcasts m', a totally ordered protocol will guarantee only that processes receive m'.
FIFO or Causal Ordering can be combined with Total Ordering if required.

All reliable multicasts have the following three properties.

Validity: If a correct process multicasts a message m, then all correct processes eventually receive m.
Agreement: If a correct process receives a message m, then all correct processes eventually receive m.
Integrity: For any message m, every correct process receives m at most once and only if m was multicast by the sender of m.

The protocols only differ in the strength of their message delivery order requirements.

Multicast Algorithms

In the algorithms to follow, R stands for Reliable Multicast, F for FIFO, C for Causal and A for Atomic.
In an asynchronous system where a reliable link exists between every pair of processes, the algorithm below demonstrates how a **Reliable multicast** can be achieved.

Every process p executes the following:-

```
multicast(R, m):
  tag m with sender(m) and seq#(m)
  send(m) to all group including p
```

The receive(R, m) occurs as follows:

```
  upon arrival(m) do
    if p has not previously executed receive(R, m) then
      if sender(m) <> p then
        send(m) to all group
        receive(R, m)
```

It is easy to use Reliable Multicast to build a **FIFO Multicast algorithm**. To F-multicast a message m, a process q simply R-multicasts m. The difference is at the receiver which orders the delivery.

```
multicast(F, m):
     multicast(R, m)
```

If m is the ith message from q, then m is tagged sender(m)=q and seq#(m)=i.
For each q, every process p maintains a counter next[q] that indicates the sequence number of the next F-multicast from q that p is willing to F-deliver. Incoming messages are placed in a message bag from which messages that can be FIFO delivered (according to the value of next[q]) are removed.

Every process p initialises:-

```
msgbag = Æ
next[q] = 1 for all q
```

The receive(R, m) occurs as follows:

```
upon arrival(R, m) do
  q := sender(m);
  msgbag := msgbag È {m};
  while ($ m'Î msgbag: sender(m')=q and seq#(m')=next[q]) do
    receive(F, m')
    next[q] = next[q]+1;
    msgbag := msgbag - {m'}
```

A **Causal multicast** algorithm can be constructed on top of a FIFO multicast.
To C-multicast a message m, a process p uses the FIFO multicast algorithm to F-multicast the sequence of messages that p has causally received since its previous C-broadcast followed by the message m. A process q receives all of the messages in the sequence only which it previously has not received.

Initialisation:

```
prevReceives = ^
```

```
multicast(C, m):
  multicast(F, <prevReceives || m>)   /* || is list concatenation operator */
  prevReceives = ^
```

The receive(C, m) occurs as follows:

```
upon arrival(F, <m1,m2,…mn>) do
  for i := 1 to n do
    if p has not received mi then
      receive(C, mi)
      prevReceives := prevReceives || mi;
```

One of the fundamental results about fault-tolerant distributed computing is the impossibility of achieving consensus in asynchronous systems which suffer crash failures. This is primarily due to the fact that it is impossible to distinguish between a process that has crashed and one that is extremely slow. The consensus problem can be easily reduced to implementing atomic multicast. For example, to propose a value, a process A-multicasts it. To decide on a value, each process picks the value of the first message that is A-received. By total order of atomic broadcasts, all processes agree on the same value. Consensus and Atomic Multicast are therefore equivalent in asynchronous systems with crash failures and so there are no deterministic Atomic Multicast protocols (total ordering protocols) for such systems.
Atomic multicast can be deterministically implemented in synchronous systems where upper bounds on message latency exist.

If we assume no failures, the algorithm given below can be used to implement a **totally ordered** multicast.

Each site maintains a 'local clock'. A clock doesn't necessarily have to supply the exact time, but could be implemented simply by a counter which is incremented after each *send* or *receive* event that occurs at the site, so that successive events have different 'times'. The algorithm executes in two phases. During the first phase the originator multicasts the message to all destinations and awaits a reply from each. Each receiver queues the message and assigns it a proposed timestamp based on the value of its local clock. This timestamp is returned to the originator. The originator collects all the proposed timestamps for the multicast and selects the highest. During the second phase of the algorithm, the originator commits the multicast by sending the final chosen timestamp to all destinations. Each receiver then marks the message as deliverable in its queue. The message queue is ordered on the value of the timestamps associated with each message each time a timestamp is updated. When a message gets to the top of the queue and is deliverable it may be delivered immediately to

the application.

As an example, consider three simultaneous multicasts, m_1, m_2 and m_3 received in arbitrary order by three sites 1, 2 and 3. The local time at Site1 is initially = 15, the local time at Site2 is initially=16 and the local time at Site3 is initially = 17. To make timestamps unique they can be generated by concatenating the local time at a site with the unique name of the site.

Initially the messages arrive in the following order at the sites:

m_3	m_2	m_1
15.1	16.1	17.1
u	u	u

m_2	m_1	m_3
16.2	17.2	18.2
u	u	u

m_1	m_3	m_2
17.3	18.3	19.3
u	u	u

Each site returns the timestamps it assigned the respective messages to their originators.
Let's say the originator of m1 is the first to get all replies and issues a commit with timestamp = 17.3.

m_3	m_2	m_1
15.1	16.1	17.3
u	u	d

m_2	m_1	m_3
16.2	17.3	18.2
u	d	u

m_1	m_3	m_2
17.3	18.3	19.3
d	u	u

Message m1 is now deliverable at Site3.
If a commit for message m2 should arrive at the sites with timestamp = 19.3 we get:-

m_3	m_1	m_2
15.1	17.3	19.3
u	d	d

m_1	m_3	m_2
17.3	18.2	19.3
d	u	d

m_3	m_2
18.3	19.3
u	d

Message m1 is now deliverable at Site2.
If a commit for message m3 should arrive at the sites with timestamp = 18.3 we get:-

m_1	m_3	m_2
17.3	18.3	19.3
d	d	d

m_3	m_2
18.3	19.3
d	d

m_3	m_2
18.3	19.3
d	d

Consider the costs of achieving totally ordered message delivery:-

Latency Cost: Each message which arrives at a site must wait in a queue for a commit and until all messages with earlier timestamp in the queue are delivered.

Bandwidth Cost: The protocol involves the exchange of several messages:- The original multicast, all the individual timestamp messages returned by the recipients and the final commit multicast.

Storage Cost: Messages are stored in a queue from time of arrival until time of delivery. This time may be longer if reliable delivery is required to all destinations as a site may have to take the role of initiator.

The causal multicast generally incurs less latency cost as messages can be delivered without having to wait for a commit. The size of each message may be larger due to the forwarding of causal dependent context with the message. The effect of this is not too severe however as with most systems it is often the additional number of messages and not their size which is the dominating cost of transmission.