Group Communication

CMPS 4760/6760: Distributed Systems
Overview

- Distributed Mutual Exclusion (15.2)
- Leader Election (15.3)
- Group communication (6.2, 15.4)
- Consensus (15.5)
Communication Forms

- Multicast: message sent to a group of processes
- Broadcast: message sent to all processes
- Unicast: message sent to a single process
Multicast vs. Unicast

- Much than a convenience for the programmer
- More efficient use of bandwidth, minimizing the delay
  - Each message sent no more than once over any communication link
  - a distribution tree + hardware multicast support

Source tree  
Shared tree  
Rendezvous point
Multicast vs. Unicast

- Example: sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto
  (a) by two separate UDP sends
  (b) by a single IP multicast operation: a single copy sent from London to a router in Palo Alto, followed by a hardware multicast via the Ethernet to destinations
Group Communication

- IP Multicast
  - Unreliable multicast
  - Weak membership management

- Group Communication
  - Reliability and ordering guarantees (chapter 15)
  - Membership management (chapter 18)

- Group communication vs. IP multicast is like TCP vs. IP
Applications

- Reliable dissemination of information to large number clients, e.g., in financial industry
- Support for collaborative apps, e.g., in the multiuser games
- Support for fault-tolerance strategies, e.g., consistent update of replicate data (chapter 18)
- Support for system monitoring and management, e.g., load balancing
Group Communication

- Programming Model (6.2.1-6.2.2)
- Case study: JGroups (6.2.3)
- Reliable group communication (15.4)
Programming Model

- Process Groups
  - Messages sent to the processes and no further dispatching
  - Messages are unstructured byte arrays with no support for marshalling
  - Similar to services provided by sockets

- Object Groups
  - A collection of objects (normally instances of the same class)
  - Each has a local proxy for the group
  - Group RMI: transparent mode vs. non-transparent mode
Programming Model

- Closed and Open Groups

- Overlapping and non-overlapping groups
Reliable Multicast

- **integrity**: message received is the same as the one sent and no duplicates
- **validity**: any outgoing message is eventually delivered
- **agreement**: if the message is delivered to one process, it is delivered to all processes
Ordered Multicast

- **FIFO ordering**: if a process sends one message before another, it will be delivered in this order at all processes in the group.
- **Causal ordering**: if a message happens before another message, this causal relationship will be preserved in the delivery at all processes.
- **Total ordering**: if a message is delivered before another message at one process, the same order will be preserved at all the processes.

\[ \text{Causal ordering} \Rightarrow \text{FIFO ordering} \]
An example of bulletin board

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>A.Hanlon</td>
<td>Mach</td>
</tr>
<tr>
<td>24</td>
<td>G.Joseph</td>
<td>Microkernels</td>
</tr>
<tr>
<td>25</td>
<td>A.Hanlon</td>
<td>Re: Microkernels</td>
</tr>
<tr>
<td>26</td>
<td>T.L’Heureux</td>
<td>RPC performance</td>
</tr>
<tr>
<td>27</td>
<td>M.Walker</td>
<td>Re: Mach</td>
</tr>
</tbody>
</table>

FIFO ordering

Causal ordering

Total ordering
Group Membership Management

- Membership service provides
  - Interface for membership changes
  - Failure detection
  - Notification of membership changes
  - Group address expansion
Case study: the JGroups toolkit

http://www.jgroups.org/

- Causal ordering
- Membership service
- Merging subgroups (after network partition)

Protocol Stack (a wide range of protocols can be combined)
- Message packetization
- IP multicast + UDP

Applications

Building blocks

Channel

CAUSAL
GMS
MERGE
FRAG
UDP
import org.jgroups.JChannel;

public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, null, "Fire!");
            channel.send(msg);
        } catch(Exception e) {
        }
    }
}
import org.jgroups.JChannel;

public class FireAlarmConsumerJG {
    public String await() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = (Message) channel.receive(0);
            return (String) msg.GetObject();
        } catch(Exception e) {
            return null;
        }
    }
}

Timeout; 0 means block until a message is received
JGroups – Building Blocks

- MessageDispatcher: send a message to a group and waits for some or all of the replies
- RpcDispatcher: invoke a method on all objects associated with a group
- ReplicatedHashMap: allow members in a group to share common state
- ...
Group Communication

- Programming Model (6.2.1-6.2.2)
- Case study: JGroups (6.2.3)
- Reliable group communication (15.4)
Assumptions

- Processes can fail only by crash, reliable one-to-one channels
- Static groups with known membership
- Each process is a member of at most one group
- Closed groups
Group Communication

- $\textit{multicast}(g, m)$ sends a message $m$ to all members of group $g$
  - $m.\textit{sender}$: the unique id of the process that sent it
  - $m.\textit{group}$: the unique destination group id

- $\textit{deliver}(m)$ delivers a message $m$ sent by a multicast to the calling process
  - A multicast message is not always handed to the application layer inside the process as soon as it is received at the process’s node
Reliable Multicast

- **integrity**: every correct process delivers a message at most once, only if some process in the group multicasts that message

- **validity**: if a correct process multicasts a message, it will eventually deliver it

- **agreement**: if a correct process delivers message \( m \), then all other correct processes in the group will eventually deliver \( m \)
Reliable Multicast via Reliable Unicast

\( P_i ::\)

\[
\text{var} \\
\quad \text{Received} = \{\}; \\
\text{multicast}(g, m) : \\
\quad \text{for each } q \in g, \text{send}(q, m);
\]

\( \text{receive}(m):\)

\[
\quad \text{if } (m \notin \text{Received}) \\
\quad \quad \text{Received} = \text{Received} \cup \{m\}; \\
\quad \quad \text{if}(P_i \neq m.\text{sender}) \\
\quad \quad \quad \text{for each } q \in m.\text{group}, \text{send}(q, m); \\
\quad \text{deliver } m \text{ to the application layer;}
\]

\( \)reliable unicast

Message complexity: \( O(N^2) \)
Reliable Multicast over IP Multicast

\[ P_i:: \]

\[ \begin{aligned} 
\text{var} & \quad \text{hold-back} = \{\}; \\
S &= 0; \quad \text{// seq no of sent msg} \\
R[1 \ldots N] &= \{\}; \quad \text{// seq no of received msg} \\
\end{aligned} \]

\[ \text{multicast}(g, m): \]

\[ \begin{aligned} 
\text{IP-multicast}(g, m, S, <q, R[q]>); \\
S &= S + 1; \\
\text{acknowledgements} & \quad \text{receive}(m, S, <q, R'[q]>): \\
p &= m.\text{sender}; \\
\text{if }(S == R[p] + 1) & \quad \text{deliver message}; \\
\quad R[p] = R[p] + 1; \\
\text{if }(S \leq R[p]) & \quad \text{message is discarded} \\
\text{if }(S > R[p] + 1) & \quad \text{put } m \text{ in the } \text{hold-back} \text{ queue} \\
\quad \text{send NACK to } p \\
\text{if }(R'[q] > R[q]) & \quad \text{send NACK to } p \text{ or } q \\
\text{negative} & \quad \text{acknowledgement} \\
\text{acknowledgements} & \quad \text{acknowledgements} \\
\end{aligned} \]
Reliable Multicast over IP Multicast

- **Integrity**
  - duplicate detection
  - error checking in IP-multicast

- **Validity**

- **Agreement**
  - missing message always detected if there are infinite multicast messages
  - there is always an available copy of a missing message if processes retain copies they have delivered indefinitely
Ordered Multicast

- **FIFO ordering**: if a correct process issues $multicast(g, m)$ and then $multicast(g, m')$, then every correct process that delivers $m'$ will deliver $m$ before $m'$

- **Causal ordering**: if $multicast(g, m) \rightarrow multicast(g, m')$, then any correct process that delivers $m'$ will deliver $m$ before $m'$

- **Total ordering**: if a correct process delivers message $m$ before it delivers $m'$, then any other correct process that delivers $m'$ will deliver $m$ before $m'$
Ordered Multicast

- Totally-ordered messages T1 and T2
- FIFO-ordered messages F1 and F2
- Causally-ordered messages C1 and C3

Hybrid ordering
  - FIFO-total ordering
  - Causal-total ordering
Ordered Multicast and Reliable Multicast

- Ordered multicast does not assume or imply reliability

- Hybrids of ordered and reliable protocols
  - reliable totally ordered multicast (atomic multicast)
  - reliable FIFO multicast
  - reliable causal multicast
  - ...

atomic multicast
Implement FIFO Ordering

- Our algorithm for reliable multicast over IP multicast guarantees FIFO ordering
- If we don’t need reliability:

\[ P_i:: \]

```plaintext
var

hold-back ={}
S = 0; // seq no of sent msg
R[1 ... N]; // seq no of received msg

multicast(g, m) :
B-multicast(g, m, S);
S = S + 1;
```

```plaintext
receive(m, S):

p = m.sender;
if (S == R[p] + 1)
deliver message; R[p] = R[p] + 1;
if (S <= R[p])
message is discarded
if (S > R[p] + 1)
put m in the hold-back queue;
```
Total ordering using a sequencer

First method. Basic multicast using a sequencer

{The sequencer $S$}

```plaintext
define seq: integer (initially 0)
do  receive m →
    multicast (m, seq);
    seq := seq+1;
    deliver m
od
```
Total ordering using a sequencer

\( P_i:: \)

\[
\begin{align*}
\text{var} & \quad \text{hold-back = \{\};} \\
& \quad r = 0; \\
\text{multicast}(g, m) : & \quad B\text{-multicast} (g \cup \{\text{sequencer}(g)\}, <m, id>); \\
\text{receive}(<m, id>): & \quad \text{place } m \text{ in the hold-back queue} \\
\text{receive}(m_{\text{order} = <"order", id, s}>): & \quad \text{wait until } <m, id> \text{ in hold-back queue and } s = r; \\
& \quad \text{deliver } m; \\
& \quad r = s + 1;
\end{align*}
\]

\( \text{Sequencer::} \)

\[
\begin{align*}
\text{var} & \quad s = 0; \\
\text{receive}(<m, id>): & \quad B\text{-multicast} (g, <"order", id, s>); \\
& \quad s = s + 1;
\end{align*}
\]
The ISIS algorithm for total ordering

\( P_i \) ::

\[
\text{var}
\]

\[
\text{hold-back} = \{\};
\]

\[
A = 0; \quad \text{// largest seq no seen so far}
\]

\[
P = 0; \quad \text{// largest proposed seq no}
\]

\[
\text{multicast}(g, m) : \quad \text{B-multicast (}g, <m, id>\text{)} \text{ and wait for responses}
\]

\[
a = \text{the largest proposed seq no; } \quad \text{//the next agreed seq no}
\]

\[
\text{B-multicast (}g, <id, a>\text{)};
\]

\[
\text{receive(}m, id\text{)}:
\]

\[
\text{reply with } P = \text{max}(A, P) + 1; \quad \text{//proposed seq no}
\]

\[
\text{attach } P \text{ to } m \text{ and place it in } \text{hold-back} \text{ queue (smallest seq no first)}
\]

\[
\text{receive(<}id, a\text{>)}
\]

\[
A = \text{max}(A, a); \quad \text{\text{attach } a \text{ to msg (reorder } \text{hold-back} \text{ queue if needed)}
\]

\[
\text{transfer the msg with agreed seq no and at the front of } \text{hold-back} \text{ queue to the tail of the delivery queue}
\]

\[
\text{include unique process id in } P \text{ to ensure a total order}
\]
The ISIS algorithm for total ordering
Causal ordering using vector timestamps
Causal ordering using vector timestamps

\[ P_i:: \]

\begin{align*}
\text{var} & \quad \text{receive}(<m, t>): \\
\text{hold-back} = \{\}; & \quad j = m.\text{sender}; \\
\text{VC: array}[1..N] \text{ of integer}; & \quad \text{place } m \text{ in the } \text{hold-back} \text{ queue}; \\
\text{multicast}(g, m) : & \quad \text{wait until } t[j] = \text{VC}[j] + 1 \text{ and } t[k] \leq \text{VC}[k] (\forall k \neq j); \\
\text{VC}[i] = \text{VC}[i] + 1; & \quad \text{deliver } m; \\
\text{B-multicast } (g, <m, VC>); & \quad \text{VC}[j] = \text{VC}[j] + 1; \\
\end{align*}

- Causal multicast + reliable multicast \Rightarrow \text{reliable causally ordered multicast}
- Causal multicast + sequencer-based protocol \Rightarrow \text{causally and totally ordered multicast}
Overlapping groups

- Global FIFO ordering: If a correct process issues $\text{multicast}(g, m)$ and then $\text{multicast}(g', m')$, the every correct process in $g \cap g'$ that deliver $m'$ will deliver $m$ before $m'$.

- One can define global causal ordering and global total ordering similarly.

- A simple approach to implement global ordering:
  - Multicast each message $m$ to all the processes in the system.
  - Each process either discards or delivers $m$ according to whether belongs to $\text{group}(m)$.
Multicast in synchronous and asynchronous systems

- We have described algorithms for
  - Reliable unordered multicast
  - Reliable FIFO-ordered multicast
  - Reliable causally ordered multicast
  - Totally ordered multicast
  - Causally and totally ordered multicast
  - FIFO and totally ordered multicast
Multicast in synchronous and asynchronous systems

- Can we get reliable and totally ordered multicast (atomic multicast)?
  - Yes for synchronous system
  - No for asynchronous system even with a single process crash failure
  - Equivalent to consensus with crash failures (FLP impossibility result)