CptS 464/564
Lecture 7

14th September 2004

Message Passing (2.2)

Recall that a distributed system is characterized as being one in which the processes interact only by message passing. Message passing is thus fundamental to distributed systems. Here we look at some characteristics of the message passing paradigm.

First, for message passing to work there must be an agreement on the *protocol* to be followed: as previously discussed this includes agreement on the *format and meaning* of messages.

Second, processes in a distributed system must be able to address one another. In a network such as the internet this involves each node having a unique address. In other networks, nodes may be addressable only locally or only relatively. For example, on a ring topology you might be able to speak about "my 2nd neighbor clockwise".

Message passing is built on two primitives: send(dest, msg) and (src,msg) := recv().

Problem: in many networks (including the internet), sender cannot know that the receiver has received a message: communication is unreliable. Solution: acknowledements. Notice again the importance of distinguishing the programming language interface (API) to a protocol from the protocol itself.

Message Passing API issues

- block send until ACK is received?
  - Yes, obviously, but ...
  - how long should it wait for the ACK?
  - how should the app find out if ACK was not received?
  - maybe sender should be able to send multiple messages before waiting for ACKs?
  - Poor use of available bandwidth: bandwidth-delay product

        - Consider: link with bandwidth 1Gb/s and one-way delay 10ms. Sending 1Mb messages (much larger than usually found) and waiting for each to be acknowledged before sending the next we send 50 messages/sec = 50 Mb/sec = 5% of link capacity.
    - the answer is not so obvious after all

- if don't block until ACK received?
    - maybe still have to block for local resources such as buffer space or direct access to the network
    - how does application find out that an ACK was received?
        - ask with wait_for_ack() or inquire_ack() primitive
        - asynchronous notifications (like interrupts)
    - Note that not waiting for the ACK does not mean that send() operation can return immediately: may have to wait for *local* reasons: buffer space, network availability, ...

- Is this an application issue or a middleware issue?

Message Passing Quality of Service issues

- QoS = "How?" or "How well" rather than "what" characterization of distributed interaction
- How long does it take for messages to reach their destination? What is the frequency with which messages are lost? Why are messages lost - congestion or randomly noisy channel or correlated noisy channel?
- Are messages received in the order they were sent (FIFO channel semantics)? We will later see distributed algorithms that rely on this property.
    - If not, may have to build a mechanism to ensure FIFO using sequence numbers in the message headers. The internet TCP protocol uses header sequence numbers to guarantee FIFO delivery and often is a good starting place for distributed algorithms.
    - Duelling protocols: example SunRPC. SunRPC has its own acknowledge and retransmit mechanisms. When run over top of TCP if packets are lost, disastrous performance can result: packet is lost, RPC timeout occurs but can't be sent because waiting for TCP ack that will never come. TCP timeout occurs causing retransmission at the TCP level. Lost data now arrives, but is thrown away at the RPC level and the original request is sent again. A useful answer was delivered but thrown away.

Remote Operations (2.3)

Send/Recv/Ack is insufficient: ACK guarantees only that a message was received. Client ultimately wants to know that the message was acted upon.

Request/Reply protocols have this property.

- Client sends request to server
- Server responds with an answer (if server is going to take a long time to send an answer, it may send an ACK to inform the client that the request was received and is being worked on).
- CORBA remote method invocation

API issues for Request/Reply

- blocking send (i.e. wait for reply) or not?
- Non-blocking request (also known as asynchronous invocation) may have performance benefits, but increases complexity for both client and system software
  - For the client, the API must provide a way to retrieve replies
  - for the system (think ORB), a way must be provided to match incoming reply to one of many requests that have been sent
  - Mike Dean at BBN reported that non-blocking requests represent about 5% of remote requests in their code, statically, but 95% at run-time.
  - (This is more of an issue for clients than for servers. Why?)
- Alternative: threaded client code. Each request is made in blocking style by one of many concurrent threads.

Group Communications (2.4)

So far, we have talked about point-to-point (aka unicast) communications. Another mechanism often used in distributed systems is multipoint (aka multicast) communication. Messages are sent to all *members* of a *multicast group.*

Multicast is not the same as broadcast: broadcast goes to all members of a network; multicast goes to a subset.

Multicast mechanisms may be built over a lower-level broadcast operation: e.g. broadcast on an ethernet, or over lower-level unicast operations, or both. In either case, multicast may make better use of communication resources than n point-to-point messages. (See figure).

Multicast can serve as a building block for handy distributed services such as *transparent replication:* a number of servers implement the same service without clients being aware of the fact that there are several servers.

Key services for group communicatons

- *Group Membership Service* - operations to create and join a group, query current group membership and reachability
- *Group Communication Service* - operations to send and receive messages without knowing the participant list; *closed group:* only members can send to the group; *open group:* anyone can send to the group.
- In general different permissions to join, query, send may be granted to differnet participants, so open/closed distinction is not necessarily a strong one.
- More fundamentally, group communication mechanisms are distinguished by their *reliabilty* (message delivery) properties and their *ordering* properties.

Message ordering

Key question: in what order should members of a group receive messages sent to the group? We take this up again in Section 2.7, but notice that all of the following are "reasonable" rules for delivery orders.

- Total order: all members receive messages in the same order
- FIFO order: messages from a single sender are received in the order sent, but messages from different senders may be arbitrarily interleaved
- Causal order: FIFO, plus messages from a sender are received by a receiver after any messages previously received by that sender. (need to improve this definition).
- Save for 2.7. Exercises: give examples of sending and delivery sequences that are
  - FIFO but not causal
  - Total but not FIFO
  - FIFO but not total
  - Total and FIFO but not causal
  - Causal but not total

Reliability and omissions

- unreliability (e.g. internet)
- Atomicity: either all group members receive a particular message or none do (hard)
- To tolerate omissions use ACKs or NACKs. (What are the relative advantages and disadvantages?)

Routing in multicast protocols

- Routing should: minimize the number of messages sent (counting 1 for each time a message is sent on a link)
  - Use hardware multicast in broadcast domains; use minimum-size spanning tree
- Routing should minimize delivery latency – issue: average or maximum?
  - Use minimum-delay spanning tree to minimize average latency
- Many compromises: multiple senders to same group, minimum messages count vs minimum avg. latency vs. minimum maximum latency.

Next time: Sec. 2.5 on clocks and time