7 Indirect Communication

Saturday, January 12, 2019 2:04 PM

Explain how indirect communication is accomplished through techniques like group communication, publish-subscribe systems, message queues, and shared memory approaches.

	Time Coupled	**Time Uncoupled**
Space Coupled	3-Tier RMI Remote observer (pub/sub): sender doesn't know the receivers, but the receivers have to know the sender	
Space Uncoupled	Group communication (IP multicast) Pub/sub	Message Queue Pub/sub (if combined with message queue or storage) Tuple Spaces DSM

Group Communication

- Message is sent to a group and delivered to all members
- Sender doesn't know receiver's ID
- May be implemented over IP multicast
- Use cases: dissemination of info for many clients, e.g. stock tickers, collaborative apps where events are disseminated to preserve a common user view
- Uses a single multicast operation instead of separate sends (supported by router)
    - There is a guarantee that message is delivered (no half-way cases)
- Messages usually delivered as byte arrays, without unmarshalling support

To consider

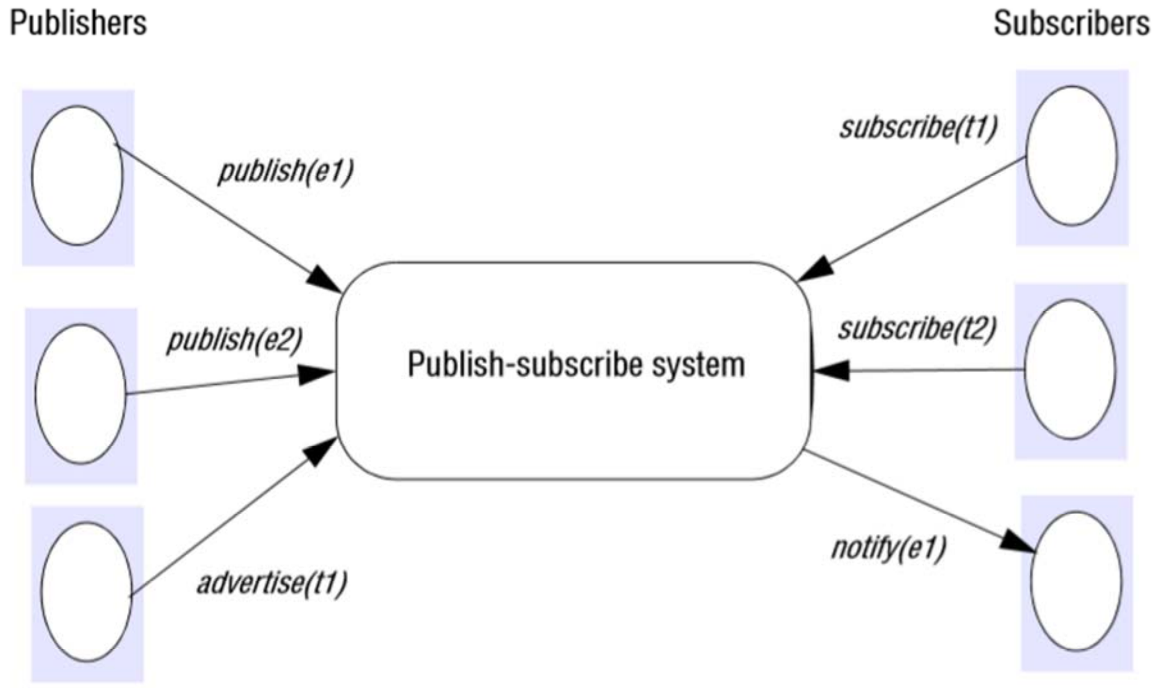
- Open/closed groups: may processes outside the group send to it?
- Non/overlapping
- A/synchronous

- Reliability
    - Integrity: m is delivered without changes, no duplicate deliveries
    - Validity: any m is eventually delivered
    - Agreement: if m is delivered to one client, it is delivered to all
- Ordering
    - FIFO
    - Causal: considers causal relationships between messages
    - Total: same delivery order for everyone

Pub Sub Systems

- Sender sends messages to a broker
- Broker notifies remote observers
- Observers can subscribe to:
    - Topic
    - Channel
    - Event

- Applications: RSS, financial info systems, monitoring, smart home



- Advertise: like channel topics on IRC

Characteristics

- Heterogeneity: subscribers can be het
- Asynchronicity: publishers & subscribers are decoupled,

Message Queues

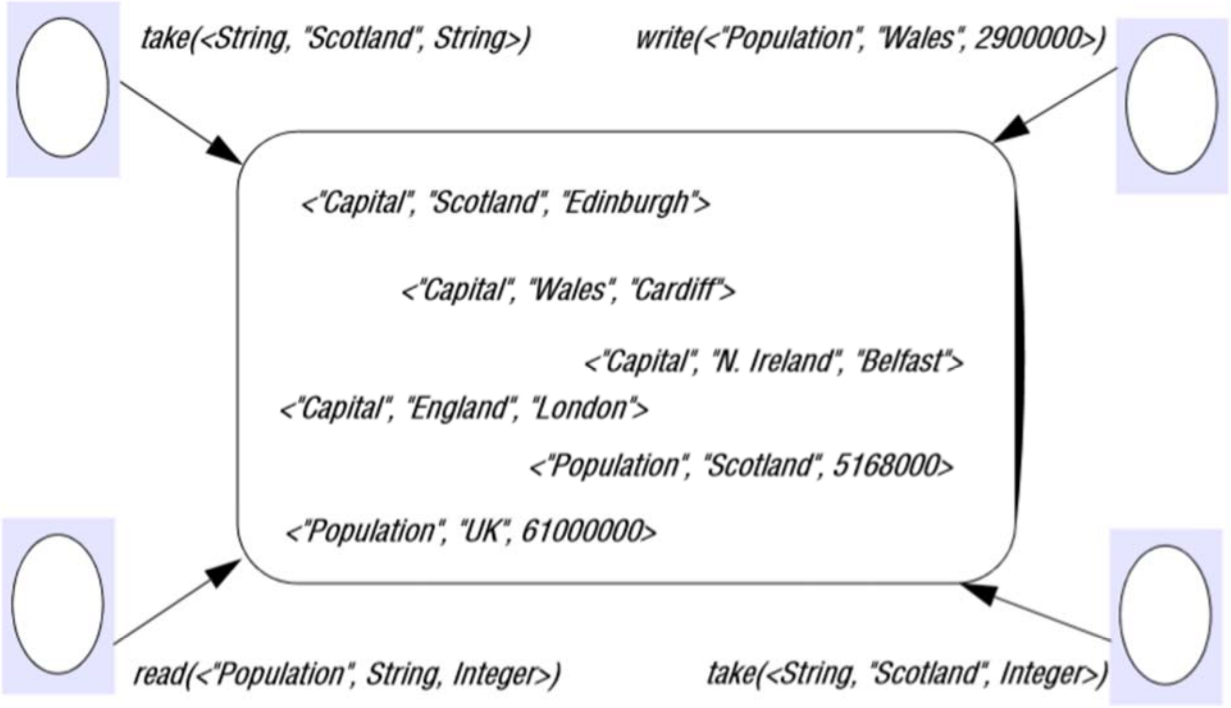
- Sender sends message to queue
- Different ways to set up connection between queue and receivers

1. Queue notifies specific receiver(s): space coupled
2. Receivers poll queue repeatedly: space uncoupled
3. Receive (blocking): the client blocks the server until the server returns a message
    - You might get network timeouts
    - This method is barely (not) used

Distributed Shared Memory

- Similar to a hashmap
- RAM that everyone can read and write on
- Reader doesn't know when the information was written, or who left it there
- For parallel systems, not so much for client-server

Tuple Spaces


take(<String, "Scotland", String>)
write(<"Population", "Wales", 2900000>)
<"Capital", "Scotland", "Edinburgh">
<"Capital", "Wales", "Cardiff">
<"Capital", "N. Ireland", "Belfast">
<"Capital", "England", "London">
<"Population", "Scotland", 5168000>
<"Population", "UK", 61000000>
read(<"Population", String, Integer>)
take(<String, "Scotland", Integer>)

- Similar to DSM BUT once you've read from the tuple space, the information is gone (read vs take?)
- Use request-response pattern to ensure that data lost on transmission is not deleted accidentally from the tuple space: server marks the data that was read as [to be deleted] but only deletes it once it has a confirmation from the receiver; marked data cannot be read by others
- Space & time uncoupled