Today

- Basics of IPC in distributed systems
- Overlays
- Models for communication – RPC, multicasting, publish-subscribe, MoM, ...

Communication
Types of communication

- **Persistent or transient**
  - Persistent – A submitted message is stored until delivered
  - Transient – The message is stored only as long as the sending/receiving applications are executing

- **Asynchronous or synchronous**
  - Sender continues (asynchronous) or blocks (synchronous) until request has been accepted
  - Points of synchronization: (1) at request submission, (2) at request delivery or (3) after processing

- **Reliability**
  - Integrity – message arrives uncorrupted and without duplication
  - Reliability – message arrives despite a reasonable # of packet drops

- **Ordering** – For some applications, out of order = failure

- **Discrete or streaming**
  - Each message is a complete unit of information or part of whole
API of Internet protocols

- Both UDP and TCP use the socket abstraction (endpoint for communication)

- UDP
  - Messages are sent without acks or retries
  - Too large messages are truncated on arrival
  - Typically send is asynchronous and receive synchronous
  - Messages may be drop or delivered out of order

- TCP
  - Application can ignore message size
  - Messages are ack for reliability
  - TCP will try to match the speed of the processes reading from/writing to the stream
Data representation and marshalling

- Processes’ information is kept as data structures
- Marshalling/unmarshalling – assembling/disassembling a set of data for transmission
  - Data structures must be flattened before transmission and rebuilt after
  - Client and server may have different data representations
    - Client and server have to agree on encoding
      - How are basic data values represented (integers, floats, …)
      - How are complex data values represented (arrays, unions)
  - Both need to properly interpret message to transform them into machine-dependent representations
- Some alternative approaches
  - Sun’s XDR and Corba CDR
  - Java’s object serialization
  - XML
Remote Procedure Call (RPC)

- Earliest and best known example of a more programmer friendly model
- Some observations
  - Developers are familiar with simple procedure model
  - Well engineered procedures operate in isolation
  - There’s no fundamental reason not to execute procedures on a separate machine
- Can you hide sender/receiver communication using procedure calls?

![Diagram](image-url)
Basic RPC operation

A RPC occurs in the following steps:

1. Client procedure calls client stub
2. Client stub builds msg.; calls local OS
3. OS sends msg. to remote OS
4. Remote OS gives msg. to stub
5. Stub unpacks parameters and calls server

6. Server returns the result to stub
7. Server stub packs it in a msg. and calls local OS
8. Server’s OS sends msg. to client’s OS
9. Client’s OS gives msg. to client stub
10. Stub unpacks result and returns to client
RPC details

- How do you bind to the server? Type and instance of an interface
- How do you do parameter marshalling?
- Parameter passing and global variables
  - Copy in/copy out semantics – while procedure is being executed, nothing can be assumed about parameter values
  - All data to be worked on is passed by parameters; no reference to global data
- How about pointers?
  - Copy/restore instead of call-by-reference
  - Remote reference for more complex structures
- How do you generate client/server stubs?
- How do you transport the arguments/results?
- Simple calls – optimizing for the common case
- ...
- What does the client do after the call?
Asynchronous RPCs

- Get rid of the strict request-reply behavior, but let the client continue w/o waiting for server’s answer.
Deferred synchronous RPCs

- A variation – Client can also do a (non)blocking poll at the server to see whether results are available
RPCs with geo-distributed services

- Distributed enterprise applications built from basic, geo-distributed services
  - e.g. Hotmail: email storage, address book services, authentication service, virus scanning

- Traditional RPC
  - Undesirable, costly communication patterns \((A \rightarrow B \rightarrow A)\)
    - A calls B which returns to A
      - B calls E which returns to B
      - B calls F which returns to F
    - A calls C which …

\[ \text{Figure 1: (Left) Standard RPCs. (Right) RPC chain.} \]
Chain of RPCs

- Idea – Come back once you are done with everything
  - Embed chaining logic as part of RPC call (*chaining functions*)
  - Request flows from server to server without client interference
  - Form \( A \rightarrow B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow A \)

- Benefits
  - Fewer communication hops
  - Lower end-to-end latency
  - Potentially lower bandwidth consumption

- Assumption
  - Isolation from bugs yes, but trusted environment within a single admin domain and little churn
RPC chain execution

- **Chaining function**
  - Stand alone code (C#) specifying next service function to run – dynamic unfolding
  - Can be nested at multiple levels (use a chain stack)

- **RPC chain execution**
  - Client calls RPCC with sever, reference to a service function, parameters and chaining function
  - Server runs service function and passes results to chaining function
  - Service function returns next server, service function and chaining function
  - Function migration is through source code repository
  - Broken chains detected through an end-to-end heartbeat timeout
    - Recovery – chains, as regular RPC are idempotent
  - Exceptions propagate up the stack and back chain
Gossip-based data dissemination

- Model after the spread of epidemics
- Applications
  - Update propagation or information aggregation
- Assuming there are no write-write conflicts
  - Update operations initially performed at one (few) nodes
  - Node passes its updated state to a limited set of neighbors
  - Update propagation is lazy, eventually each update should reach every node
Gossip-based data dissemination

- Propagation models
  - Anti-entropy
    - Node chooses another at random, and exchanges differences
    - Push, pull or push/pull
  - Gossiping
    - Node just updated, tells others about it; if the node contacted already knows about it, the source stops w/ probability $1/k$
    - If you need everyone to know, gossiping along doesn’t do it

- And how do you delete items?!
  - Death certificates and dormant death certificates
Group communication – multicast

- A key service for many interesting applications
  - Online gaming, video conferencing, content distribution …
- Multicast
  - Decouples # of receivers from amount of state kept at nodes
  - Reduces redundant network communication
- Approaches to group communication

**IP Multicast**
- Needs router support
- Scalability problems (e.g. per group state)
- Network management issues

**Basic unicast**
- Scalability issues with replication at source, link stress, …
Application level multicast - issues

- **Overlay Multicast**
  - Multicast functionality pushed to end systems
  - End hosts use unicast for distribution of multicast messages

- **Some issues**
  - Minimize
    - Link stress – how often does a msg cross the same physical link
    - Stretch – delay bet/ overlay and network-level path
  - Churn, i.e. high transiency of end systems
  - Root-bottleneck problem for bandwidth-intensive applications
  - Uneven load distribution of tree-based protocols
Peer-to-Peer systems

- Demand for Internet services is expected to grow limited only (?) by the size of human population
- P2P – enabling resource sharing at that scale
  - Leveraging data and available resources of participants
  - Without separately managing servers and associated infrastructure
- Goal is to deliver a service that is self-organizing and dynamically manages load between participants
  - Algorithms for placement and retrieval are key
  - Computers and network connections are volatile
- Several antecedents (e.g. DNS, Netnews/Usenet and Grapevine), but the potential emerged only with the growth of broadband
P2P generations

- **Napster ~ launched 1999**
  - Showed the feasibility of the model, even with simple ideas (e.g., hope based biased connections, centralized indexing)

- **File sharing with increased scalability, anonymity, and fault tolerance (unstructured)**
  - E.g., Freenet, Gnutella, Kazaa, BitTorrent

- **P2P middleware (mostly structured)**
  - E.g., Pastry, Chord, Tapestry, CAN, Kademlia)
  - Place resources on a participant computers and route requests to them in a bounded number of hops
  - Resources are identified by GUID, usually based on a hash of some of the state

- **Structured or unstructured P2P?**
  - Search efficiency (in number of hops) and overhead or self-organizing and expensive, probabilistic search
## Space and time coupling

<table>
<thead>
<tr>
<th></th>
<th>Time-coupled</th>
<th>Time-uncoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space coupling</strong></td>
<td>Communication directed to a given receiver(s) that must be available at the time e.g. messaging passing, RPC/RMI</td>
<td>Sender(s) and receiver(s) can have independent lifetimes e.g. Mailbox</td>
</tr>
<tr>
<td><strong>Space uncoupling</strong></td>
<td>Sender does not need to know ID of receiver but they must exist at the same time e.g. IP multicast</td>
<td>Sender does not need to know ID of receiver; sender(s) and receiver(s) can have independent lifetimes e.g. message oriented middleware</td>
</tr>
</tbody>
</table>
Publish-subscribe

- Publishers publish structured events; receivers subscribe to them
- Some variants
  - Channel-based – publishers publish events to a named channel; primitive but still useful
  - Topic- or subject-based – close to channel-based but topics are explicitly defined; one can build hierarchies of topics
  - Content-based – generalization of topic-based allowing the expression of subscriptions over a range of fields
  - Type-based – based on type of events; matching can be done based on types or subtypes
Architecture of publish-subscribe systems
Message Oriented Communication

- Point-to-point comm. through an intermediary queue
- Asynchronous persistent communication through support of middleware-level queues – queues correspond to buffers at communication servers
- Common primitives
  - Put – append a msg to a given queue
  - Get – Block until the specified queue is non-empty and remove the first msg
  - Poll – Check a given queue for message and remove first, never block
  - Notify – Install a handler to be called when a msg is put into the given queue
- Queuing of msgs. can be FIFO or priority-based
- E.g. IBM WebSphere MQ, Java Messaging Service
Message brokers and apps. integration

- Message queuing systems assume a common messaging protocol: all applications agree on message format
- To use MQ systems for integration – message broker: takes care of application heterogeneity
  - Transforms incoming messages to target format
  - Often acts as an application gateway
  - May provide subject-based routing capabilities
Shared memory approaches

- **Distributed shared memory**
  - Access appears to be to a shared address space
    - Done by underlying runtime system using msg. passing
  - Reading and writing bytes
  - Accessed by address

- **Tuple space**
  - First introduced with *Linda* by D. Gelernter now adopted by IBM Tspaces, JavaSpaces, etc.
  - Higher-level than DSM; semi-structured data
  - Associative access – write into a tuple space and read or take from it
Summary

- Communication is at the heart of distributed systems
- Powerful primitives makes programming them a lot easier
- Solutions for large distributed systems should consider a number of different issues
  - Referential and temporal decoupling
  - Group communication
  - ....