Remote Invocation

Today
- Request-reply, RPC, RMI

Next time
- Overlay networks and P2P
Types of communication

- Persistent or transient
  - Persistent – A submitted message is stored until delivered
  - Transient – … stored as long as the sending/receiving app is 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 – msg arrives uncorrupted and without duplication
  - Reliability – msg arrives despite some packet drops

- Ordering

- Discrete or streaming
API of Internet protocols

- 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
  - Message identifiers are associated with each packet so recipient can detect duplicate and reorder if necessary
  - 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 but sent in messages as sequence of bytes
- Marshalling/unmarshalling – assembling/disassembling process’ data for transmission
  - Data structures must be flattened before xfer 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’ XDR and Corba CDR
  - Java’s object serialization
  - XML – a textual format for representing structured data
Request-reply protocols

- *Find for sensing messages, but how do you get something done?*
- Request-reply – supports client/server msg exchange
- Normally, synchronous (client blocks) and reliable
- Described based on three primitives
  - `doOperation` – used by client to invoke operation
    - Args specify remote server and arguments; result is a byte array
    - Client is assumed to marshal/unmarshal the arguments/response
    - After sending it, client issues a receive to get the reply
  - `getRequest` – used by server to get request
  - `sendReply` – used by server to send reply
    - When received by the client the original `doOperation` is unblocked
Protocols and usage

- Message id – for reliable delivery, for instance, need a msg identifier: request id + sender id
- Failures
  - Omission – server omits to respond to a request
  - Out of order delivery
  - Process failure (crash) or lost messages – timeout
  - Duplicate request – check identifier, support idempotent operations
- Style of exchange protocols

<table>
<thead>
<tr>
<th>Name</th>
<th>Client</th>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Request</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
<td>Reply</td>
<td></td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
<td>Reply</td>
<td>Ack reply</td>
</tr>
</tbody>
</table>

- HTTP – an example request-reply protocol
Remote Procedure Call (RPC)

- Earliest and best known example of a more programmer friendly model (Birrell and Nelson in 1984)
- 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 of Remote Procedure Call](image)
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

Client machine

- Client process
  - \( k = \text{add}(i, j) \)
  - \( \text{proc: "add"} \)
  - \( \text{int: val}(i) \)
  - \( \text{int: val}(j) \)

Server machine

- Server process
  - \( k = \text{add}(i, j) \)
  - \( \text{proc: "add"} \)
  - \( \text{int: val}(i) \)
  - \( \text{int: val}(j) \)
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?
  - RPC gen based on IDL spec
- Several possible semantics – maybe, at-least-one, at-most-one
- ...
- *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

- Combining two asynchronous RPC is sometimes also referred to as deferred synchronous RPC

- A variation – Client can also do a (non)blocking poll at the server to see whether results are available
Remote Method Invocation

- RMI extends RPC into the world of distributed objects
  - As RPC, support programming with interfaces
  - ... Is built on top of request-reply and different call semantics
  - ... and offer similar level of transparency

- But
  - Programmer can use power of OO programming (objects, classes, inheritance ...)
  - All objects (local or remote) have an object reference; these references can be passed on as parameters

- Distributed objects
  - The state of an object is logically partitioned and so easily distributed (they could also be replicated and/or migrated)
  - With distributed objects, distributed garbage collection
    - One way to implement it – cooperating local collectors
Summary

- Communication is at the heart of distributed systems
- Powerful primitives can make programming them a lot easier
- We discussed three paradigms for distributed programming
  - Request-reply
  - RPC
  - RMI
- In the next few weeks, overlays and P2P, and indirect communication for additional help