Remote Invocation

Today
- Request-reply, RPC, RMI

Next time
- Indirect communication
Data representation and marshalling

- Processes’ information
  - kept as data structures but sent in msgs 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)
    - Agreed upon intermediate language or source’s representation
      - Both need to properly interpret message to transform them into machine-dependent representations

- Multiple external representation alternatives
  - Sun’s XDR, Corba, XML, Google’s protocol buffer, JSON
Beyond message passing

- At the core of distributed systems – message passing
- Send/receive exposes communication
- Are there programmer-friendlier models?
  - Request-reply – a pattern on top of message passing with little support for request/reply interactions
  - Extending procedure calls – remote procedure calls
  - Working with remote objects – remote method invocation
Request-reply protocols

- 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 client, original `doOperation` is unblocked

Request message
- Select operation
- Execute operation
- `sendReply`
Request-reply protocols

- Supports client-server type interaction
- Normally, synchronous (client blocks) and reliable
  - Asynchronous is also possible
- Synchronous and asynchronous
  - 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
Request-reply protocols

- For reliability
  - Messages need a requestId + a process identifier
- Failures partially depend on transport
  - TCP or UDP
- To handle omission failures – timers
  - For duplicate messages, msg id or idempotent operations or history
- Exchange styles – different behavior in front of failures
  - *Request* (*R*) – Used when client don’t need confirmation, asynchronous (typically over UDP)
  - *Request-reply* (*RR*) – Client/server
  - *Request-reply-ackReply* (*RRA*) – let server clean history

<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 as an example

- HyperText Transfer Protocol
  - Specifies msgs exchanged, formats, methods …
  - Content negotiation – clients state format they can accept
  - Password-style authentication

- Connections
  - Client interaction in version 1.0
    - Client request a connection at default (or given) port
    - … sends request msg to server
    - Server sends reply
    - Connection is closed
  - Setting/closing a connection per request is costly
    - HTTP 1.1 uses persistent connections
    - Can be close by client, server or after being idle for a while

- Msgs marshalled into ASCII text strings
HTTP as an example

- Clients invoke methods to be applied to resources at the server (given by the URL)
- Some known methods
  - GET – Requests resource or run program pointed to by URL
  - HEAD – Same as GET but returns only metadata
  - POST – Provides data, depending on function supported by the program specified by the URL (e.g., posting a msg)
  - PUT – Requests to store data with the given URL as ID
HTTP as an example

Using telnet

```bash
$ telnet www.golang.org 80
Trying 64.233.191.141...
Connected to golang.org.
Escape character is '^]'.
GET /index.htm HTTP/1.1
host: www.golang.org

HTTP/1.1 302 Found
Location: http://golang.org/index.htm
Date: Thu, 09 Apr 2015 14:21:32 GMT
Content-Type: text/html; charset=UTF-8
Server: Google Frontend
Content-Length: 224
Alternate-Protocol: 80:quic,p=0.5

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
Remote Procedure Call (RPC)

- Earliest and best known example of a more programmer friendly model [Birrell and Nelson ’84]
- 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 hide sender/receiver comm using proc calls?
Basic RPC implementation and operation

Client machine

(1) Client calls stub

(2) Stub builds msg, calls OS

(3) OS sends msg to remote OS

Client process

k= add(i,j)

proc: “add”
int: val(i)
int: val(j)

Client stub

Server machine

(6) Server returns result to stub

(7) Packs and calls OS

(8) OS sends msg back

Server process

k= add(i,j)

proc: “add”
int: val(i)
int: val(j)

Server stub

(9) Client OS gives msg to stub

(10) Unpacks results and return

Client OS

int: val(i)
int: val(j)

Server OS

Implementation of add

int: val(i)
int: val(j)

Client machine

(4) Remote OS gives msg to stub

(5) Stub unpacks args and calls server
RPC details

- RPC promote programming with interfaces
  - Better abstractions, language independence, better maintainability
  - Interface specification with lang independence – Interface Definition Languages (e.g., XDR, Google’s protocol buffer)

- RPC, local procedure calls and transparency
  - 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
  - Failures and latency
RPC details

- Runtime is given
  - RPCRuntime was part of Cedar in the original RPC system
- Programmers writes client and server
- Client and server-stub are user generated
  - Based on the interface specification
  - By *Lupine* in the original
- A binder for clients to find where to connect (port mapper in Sun RPC)
  - Binder runs on a well-known-port
  - Manage table of references/ports for each service
RPC details

- RPC call semantics
  - Depending on fault tolerance measures:
    - Retransmit request until getting a reply or decide server failed
    - Duplicate filtering at the server
    - Re-execute procedure or retransmit reply, keeping history of results at the server

<table>
<thead>
<tr>
<th>Call semantics</th>
<th>Fault tolerance measures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Retransmit request</td>
</tr>
<tr>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>At-least-once</td>
<td>Yes</td>
</tr>
<tr>
<td>At-most-once</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- What are the semantics of local procedure calls?
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 OO programming features (objects, classes, inheritance …)
  - All objects (local or remote) have an object reference; these references can be passed on as parameters
Remote Method Invocation

- Object state is logically partitioned, easily distributed
  - With a client/server model; objects managed by servers and clients invoking them through RMI
  - Objects could also be replicated and/or migrated

- With distributed objects, distributed garbage collection
  - One way to implement it – cooperating local collectors
    - Server keeps list of processes holding remote refs to its objects
    - When a client first receive a remote reference to a particular remote object, adds itself as holder at server (*extra invocation*)
    - When client garbage collects proxies for remote objects, first removes itself from holders at server (*extra invocation*)
  - Keeping resources at servers and leases
    - What to do if clients go away? Set up leases granting the use of resources for a fixed period of time
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
- Next time – indirect communication