Remote Invocation

To do ...

- Today
- Request-reply, RPC, RMI
- Next time: Indirect communication
All IPC in distributed systems is based on low-level message passing
  – As we discussed in the last two lectures

A bit too low level
  – Modern distributed systems can have $10^3$-$10^6$ processes scattered around
Beyond message passing

- Need a higher-level of abstraction – send/receive exposes communication
- Are there programmer-friendlier models?
  - Today …
  - Request-reply – patterns in msg passing with little support for request/reply interactions
  - Remote procedure calls – extending procedure calls
  - Remote method invocation – … to remote objects

Client → Server

Do something → … Working

There you go
Request-reply protocols

- We can describe it based on three primitives
  - `doOperation` – used by client to invoke operation
    - Args specify remote server and arguments; result is a byte array
    - 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-reply protocols

- 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, two concerns
  - Integrity – msg arrives uncorrupted and without duplication
  - Reliability – msg arrives despite some packet drops
Request-reply protocols

- For reliability or request-reply communication
  - Messages need a requestId and a process identifier

- Failures partially depend on transport
  - TCP or UDP
  - Over UDP, omission and out-of-order issues
  - Process may also fail (crash failures)

- To handle omission failures – timers
  - For duplicate messages, msg id (keep a history) or idempotent operations

```
messageType: int (req/reply)
requestId: int
remoteReference
operationId: int
arguments: byte[]
```
Request-reply protocols

- Exchange styles – different behavior in front of failures
- *Request (R)* – When client doesn’t need confirmation, asynchronous (typically over UDP)
- *Request-reply (RR)* – Useful for most client-server exchanges
  - No need special ack, server reply is an implicit ack
- *Request-reply-acknowledge Reply (RRA)* – Server can 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>
Using TCP or UDP to implement RR

- Client-server exchange can be built on UDP or TCP (or any other transport, of course)
- To avoid implementing multi-packet protocols, TCP
  - TCP reliability means no need for retransmissions, duplicate filtering or history
  - No problem with large transmissions, flow-control handles it
  - If multiple exchanges, connection overhead applies once
- If you can live without all this, maybe a more efficient protocol over UDP
  - Sun NSF transmits fixed-size blocks between client/server
  - All operations are idempotent, so no need for history
Data representation and marshalling

- Processes keep information in data structures
  - records, arrays, strings, trees …

- But IPC is in msgs, sequences of bytes
  - TCP/UDP gives the mechanisms to send sequences of bytes
  - Processes need a protocol to make the exchange meaningful
  - To serialise this data into a streamy of bytes to write it, and deserialise it to read it: marshalling and unmarshaling
Data representation and marshalling

- Marshalling/unmarshalling
  - Assembling/disassembling process’ data for transmission
  - Client and server may have different data representations
  - Both need to properly interpret msg to transform it into machine-dependent representation
    - Agree on encoding
      - How are basic data values represented (integers, floats, …)
      - How are complex data values represented (arrays, unions)
    - Intermediate language or source’s representation

- Multiple external representation alternatives
  - Sun’s XDR, Corba, XML, ASN.1, Google’s protocol buffer, JSON, Gob (Go specific)
An example with ASN.1*

```go
package main

import (  
    "bytes"  
    "encoding/asn1"  
    "net"  
    "os" ...
)

func main() {
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
        os.Exit(1)
    }
    service := os.Args[1]
    conn, err := net.Dial("tcp", service)

    result, err := readFully(conn)

    var newtime time.Time
    _, err1 := asn1.Unmarshal(result, &newtime)


    os.Exit(0)
}
```

*Complete example in https://jan.newmarch.name/go/serialisation/chapter-serialisation.html*
HTTP as an example

- HyperText Transfer Protocol, on top of TCP
  - Specifies msgs exchanged, formats, methods, arguments and results, representation for marshaling …
  - Content negotiation – clients state format they can accept
  - Password-style authentication

- A fixed set of methods, some well-known ones
  - 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
  - Others: DELETE, OPTIONS, TRACE
HTTP as an example

- Clients invoke methods to be applied to resources at the server (given by the URL)
- Msgs marshalled into ASCII text strings
- 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
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
</body></html>
```

Using go

```go
package main

import (  
    "net/http" ...
)

func main() {
    if err != nil { ...
        defer response.Body.Close()
        contents, err := ioutil.ReadAll(response.Body)
        if err != nil { ...
            fmt.Printf("%s\n", string(contents))
        }
}
```
Back in 5’

- … RPC and RMI
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
  - No fundamental reason not to execute procedures on a separate machine

- Can hide sender/receiver comm. using proc calls?
RPC details

- RPC promote programming with interfaces
  - Better abstractions & maintainability, language independence
  - Interface specification with lang independence – Interface Definition Languages (e.g., XDR, Corba IDL)

- 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 ref to globals
  - How about pointers?
    - Copy/restore, no call-by-reference
    - Remote reference for more complex structures
  - Failures and latency
- **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?**
Basic RPC implementation and operation

Client machine

Client process

k = add(i, j)

proc: “add”
int: val(i)
int: val(j)

Client OS

(1) Client calls stub

(2) Stub builds msg, calls OS

(3) OS sends msg to remote OS

Server machine

Server process

k = add(i, j)

proc: “add”
int: val(i)
int: val(j)

Server stub

Implementation of add

(4) Remote OS gives msg to stub

(5) Stub unpacks args and calls server

(6) Server returns result to stub

(7) Packs and calls OS

(8) OS sends msg back

(9) Client OS gives msg to stub

(10) Unpacks results and return
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
  - Binder runs on a well-known-port
  - Manage table of references/ports for each service
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
Sun RPC

- Defined in RFC 1831
- Designed for client-server communication in the Sun Network File System (NFS)
- Run over UDP or TCP, using at-least-once semantics
- Sun XDR as external data representation and IDL
- Interfaces are identified by program and version number
- Binder (port mapper) for clients to find where to connect
- Authentication through fields in the request/reply msgs
Remote Method Invocation

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

- But
  - Programmer can use OO programming features (objects, classes, inheritance ...)
  - All objects have an object reference; refs can be passed on as parameters (first class values)
    - Not just parameter passing by value, good for complex parameters
  - For distributed objects, remote object refs and remote interfaces
Remote Method Invocation

- With OO, state partition among processes as objects
- If using a client-server model,
  - Objects managed by servers, invoked by clients through RMI
  - Objects could also be replicated and/or migrated for reliability, availability or performance

- Implementing RMI
  - Similar to RPC, a proxy object, two communication modules and a dispatcher & skeleton

- With distributed objects, distributed garbage collection
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 ref. to a remote object, adds itself as holder at server (*extra invocation*) and creates a proxy
    - Server adds clients to holders
  - When client garbage collects proxies for remote object, removes itself from holders at server (*extra invocation*) then deletes proxy
    - Server removes client from holders
  - Java’s approach

- 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

- Powerful primitives can make (distributed) programming them a lot easier
- Procedure calls
  - Simple way to pass control and data
  - Elegant and transparent way to distribute applications
  - Not the only way
- Hard to provide true transparency
  - Failures, performance, memory access, …
- How to deal with hard problems – let the programmer do it – “worse is better” (Richard Gabriel’s)