Remote Invocation

Todo ...

- Request-reply, RPC, RMI
- Next time: Indirect communication
In distributed systems, all IPC is based on low-level message passing

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

- Programmer-friendlier models
  - Request-reply – patterns in message passing with little support for request/reply interactions
  - Remote procedure calls – extending procedure calls
  - Remote method invocation – … to remote objects
We can describe them based on three primitives
- `doOperation` – client to invoke an op at the server
  - Parameters specify remote server and arguments
  - After sending it, client issues a receive to get the reply
- `getRequest` – server to get request
- `sendReply` – server to send reply
  - When client gets it, original `doOperation` is unblocked
RR protocols – Synch/Asynch and reliable

- Normally, synchronous and reliable
  - Asynchronous is also possible

- Synchronous and asynchronous
  - Synchronous – sender blocks until request is accepted
    - Points of synchronization: (1) at request submission, (2) at request delivery or (3) after processing
    - With asynchronous the sender continues instead

- Reliability, two concerns
  - Reliability – msg arrives despite some packet drops
  - Integrity – msg arrives uncorrupted and w/o duplication
RR protocols – Synch/Asynch and reliable

- For reliability or for request-reply communication
  - Messages need a request 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 duplicates, msg id (keep a history) or
  - idempotent operations
RR protocols styles

- 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 for special ack, server’s reply an implicit ack
- *Request-reply-acknowledge (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, TCP or any other transport
- Avoid implementing multi-packet protocols, TCP
  - TCP reliability—no need for retransmissions, duplicate filtering or history
  - Large transmissions are OK, flow-control handles it
  - If >1 exchanges, connection overhead applies once
- If you can live without this, maybe a more efficient protocol over UDP
  - Sun NFS transmits fixed-size blocks bet/ client/server
  - All operations are idempotent, so no need for history
Data representation and marshaling

- Processes keep information in data structures
  - Records, arrays, strings, trees ...

- But IPC is in messages, strings of bytes
  - TCP/UDP gives the mechanisms to send sequences of bytes
  - Processes need a protocol to make the exchange meaningful
  - To serialise data into a stream of bytes and deserialise it to read it
Data representation and marshaling

- Assembling/disassembling process’ data for transmission
- Client and server may have different data representations
  - Both need to properly interpret a 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
- E.g., Sun’s XDR, ASN.1, JSON, Gob, …
package main

import {
    "bytes"
    "encoding/gob"
    "fmt"
    "log"
}

type P struct {
    X, Y, Z int
    Name    string
}

type Q struct {
    X, Y *int32
    Name string
}

func main() {
    // Initialize the encoder and decoder. Normally enc and dec would be
    // bound to network connections and the encoder and decoder would
    // run in different processes.
    var network bytes.Buffer // Stand-in for a network connection
    enc := gob.NewEncoder(&network) // Will write to network.
    dec := gob.NewDecoder(&network) // Will read from network.
    // Encode (send) the value.
    err := enc.Encode(P{3, 4, 5, "Pythagoras"})
    if err != nil {
        log.Fatal("encode error:", err)
    }
    // Decode (receive) the value.
    var q Q
    err = dec.Decode(&q)
    if err != nil {
        log.Fatal("decode error:", err)
    }
}
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

$ 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/
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>

package main
import (
    "net/http"
    ...
)
func main(){
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "Usage: %s URL\n", os.Args[0])
        os.Exit(1)
    }
    response, err := http.Get(os.Args[1])
    if err != nil {
        log.Fatal(err)
    } else {
        defer response.Body.Close()
        if err != nil {
            log.Fatal(err)
        }
    }
}
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 procedure calls?
RPC details

- RPC promotes 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 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>
<th>Retransmit request</th>
<th>Duplicate filtering</th>
<th>Re-execute/retransmit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe</td>
<td></td>
<td>No</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>At-least-once</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td>Re-exec</td>
</tr>
<tr>
<td>At-most-once</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Retransmit reply</td>
</tr>
</tbody>
</table>

- What are the semantics of local procedure calls?
Basic RPC implementation and operation

(1) Client calls stub
(2) Stub builds msg, calls OS
(3) OS sends msg to remote OS

Client machine

Client process

k = add(i, j)

proc: "add"
int: val(i)
int: val(j)

Server machine

Server process

Implementation of add

k = add(i, j)

proc: "add"
int: val(i)
int: val(j)

(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
- 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
  - Clients multicast RPCs to all port mappers specifying program and version number, the port mapper forwards calls to the appropriate local service program (if there is one)
- Authentication through fields in the request/reply msgs
  - Access control mechanisms can be built on this: UNIX style, suing a shared key for signing RPC msgs, Kerberos, …
Remote Method Invocation

- RMI extends RPC into the world of distributed objects
  - As RPC, programming with interfaces
  - … built on top of request-reply, offering 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
Distributed objects and RMI

Client invokes a method

Client OS

Client

Proxy

Same interface as object

Skeleton invokes same method as object

Server OS

Server machine

Server

Object

Methods Interface

Marshalled invocation is sent over the network

Client OS

Client machine

Skeleton

Invokes same method as object

Client

Invokes a method
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 an 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
Other RPC examples

- **XML/RPC**: Over http, huge XML parsing overhead, txt transfer
- **SOAP**: Designed for web services via HTTP, same XML overhead
- **CORBA**: Quite comprehensive and very complex
- **COM**: Mostly for Windows client software
- **Protocol Buffers**: Lightweight, developed by Google
- **Thrift**: Lightweight, fancy types (lists, maps, …), developed by Facebook
- **GOB**: Go-specific, lightweight, using reflection, efficient wire-format
Summary

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