Distributed systems
Lecture 9: Introduction to distributed systems, client-server computing, and RPC

Michaelmas 2018
Dr Richard Mortier and Dr Anil Madhavapeddy
(With thanks to Dr Robert N. M. Watson and Dr Steven Hand)
Recommended reading

- “Operating Systems, Concurrent and Distributed S/W Design“, Bacon & Harris, Addison-Wesley 2003
  – or “Concurrent Systems”, (2nd Ed), Jean Bacon, Addison-Wesley 1997
What are distributed systems?

- A set of discrete computers ("nodes") that cooperate to perform a computation
  - Operates "as if" it were a single computing system
- Examples include:
  - Compute clusters (e.g. CERN, HPCF)
  - BOINC (aka SETI@Home and friends)
  - Distributed storage systems (e.g. NFS, Dropbox, ...)
  - The Web (client/server; CDNs; and back-end too!)
  - Peer-to-peer systems such as Tor
  - Vehicles, factories, buildings (?)
Preview: Lecture 15 - Google architecture
(Or: How to treat 100,000 computers as 1 computer)

- Parallel data processing: MapReduce
- Fast data analytics: Dremel
- Web serving: GWS
- Cross-datacenter RDBMS: Spanner
- Structured storage: BigTable
- Distributed locking: Chubby
- Cluster management and scheduling: Borg / Omega
- Distributed storage: Colossus

RPCs
Concurrent systems reminder

- Foundations of concurrency: processor(s), threads
- Mutual exclusion: locks, semaphores, monitors, etc.
- Producer-consumer, active objects, message passing
- Races, deadlock, livelock, starvation, priority inversion
- Transactions, ACID, isolation, serialisability, schedules
- 2-phase locking, rollback, time-stamp ordering (TSO), optimistic concurrency control (OCC)
- Durability, write-ahead logging, recovery
- Lock-free algorithms, transactional memory
- Operating-system case study

These problems were not hard enough – distributed systems add: loss of global visibility; loss of global ordering; new failure modes
Distributed systems: advantages

• **Scale and performance**
  – Cheaper to buy 100 PCs than a supercomputer...
  – … and easier to incrementally scale up too!

• **Sharing and Communication**
  – Allow access to shared resources (e.g. a printer) and information (e.g. distributed FS or DBMS)
  – Enable explicit communication between machines (e.g. EDI, CDNs) or people (e.g. email, twitter)

• **Reliability**
  – Can hopefully continue to operate even if some parts of the system are inaccessible, or simply crash
Distributed systems: challenges

• **Distributed Systems are **Concurrent Systems**
  – Need to coordinate independent execution at each node (c/f first part of course)

• **Failure of any components (nodes, network)**
  – At any time, for any reason

• **Network delays**
  – Can’t distinguish congestion from crash/partition

• **No global time**
  – Tricky to coordinate, or even agree on ordering!
• **Middleware** helps application authors write software intended to run on more than one machine at a time.
Transparency & middleware

• Recall a distributed system should appear “as if” it were executing on a single computer
• We often call this transparency:
  – User is unaware of multiple machines
  – Programmer is unaware of multiple machines
• How “unaware” can vary quite a bit
  – e.g. web user probably aware that there’s network communication ... but not the number or location of the various machines involved
  – e.g. programmer may explicitly code communication, or may have layers of abstraction: middleware
### Types of transparency

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how a resource is accessed</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where a resource is located</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that a resource may move to another location</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that a resource may be moved to another location while in use</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that a resource may be provided by multiple cooperating systems</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that a resource may be simultaneously shared by several competitive users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of a resource</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hide whether a (software) resource is in memory or on disk</td>
</tr>
<tr>
<td>Performance</td>
<td>Hide the level of demand for a service as demand changes</td>
</tr>
</tbody>
</table>
In Distributed Systems...

• We will look at techniques, protocols & algorithms used in distributed systems
  – in many cases, these will be provided for you by a middleware software suite
  – but knowing how things work will still be useful!

• Assume OS & networking support
  – processes, threads, synchronization
  – basic communication via messages
  – (will see later how assumptions about messages will influence the systems we [can] build)

• Let’s start with a simple client-server systems
Client-server model

• 1970s: development of **Local Area Networks (LANs)**
• 1980s: standard deployment involves small number of servers, plus many workstations
  – Servers: always-on, powerful machines
  – Workstations: personal computers
• Workstations request ‘service’ from servers over the network, e.g. access to a shared file-system:

![Diagram of client-server model](image)

- Client
- File server
- Network
- Disks on which shared file system is stored

Request

Reply
Request-reply protocols

• Basic scheme:
  – Client issues a request message
  – Server performs operation, and sends reply

• Example: HTTP 1.0
  – Client (browser) sends “GET /index.html”
  – Web server loads file and returns it
  – Browser displays HTML web page
Synchrony and asynchrony

• **Synchrony** and **asynchrony** have to do with waiting
• For software, this relates to a program’s event model:
  – *Synchronous clients* block awaiting a reply
  – *Asynchronous clients* can continue work while awaiting a reply
  – E.g., a command-line fetch tool vs. an interactive web browser
• For protocols, this relates to the ability to express multiple concurrent operations within a logical connection:
  – *Synchronous protocols* require that replies be issued in the same order that requests are sent
  – *Asynchronous protocols* allow **out-of-order replies** – e.g., by tagging replies with the ID number of the request
  – E.g., SMTP (one operation at a time) vs. IMAP (tagged requests)
• We often find complex combinations of synchrony and asynchrony within a single software/protocol stack
Handling errors & failures

• **Errors** are *application-level* things => easy ;-)  
  – E.g. client requests non-existent web page  
  – Need special reply (e.g. “404 Not Found”)

• **Failures** are *system-level* things, e.g.:  
  – lost message, client/server crash, network down,…

• To handle failure, client must **timeout** if it doesn’t receive a reply within a certain time $T$  
  – On timeout, client can **retry** request  
  – (Q: what should we set $T$ to?)
Retry semantics

- Client could timeout because:
  1. Request lost
  2. Request sent, but server crashed before op. performed
  3. Request sent & received, op. performed, server crashed before reply
  4. Request sent & received, operation performed, reply sent ... but lost
  5. As #4, but reply has just been delayed for longer than T
- For **read-only stateless requests** (e.g., HTTP GET), can retry in all cases, but what if request was an order with Amazon?
  - For #1, we (probably) want to re-order... in #5 we want to wait ....?
- **Worse:** We don’t know which case it actually was!
Ideal semantics

• What we want is **exactly-once** semantics:
  – Our request occurs once no matter how many times we retry (or if the network duplicates our messages)

• E.g. add a **unique ID** to every request
  – Server remembers IDs, and associated responses
  – If sees a duplicate, just returns old response
  – Client ignores duplicate responses

• Pretty tricky to ensure exactly-once in practice
  – E.g. if server explodes ;-)}
Practical semantics

• In practice, protocols guarantee one of:
  • **All-or-nothing** (atomic) semantics
    – Use scheme on previous page; persistent log
    – (similar idea to transaction processing)
  • **At-most-once** semantics
    – Request carried out once, or not at all
    – If no reply, we don’t know which outcome it was
    – e.g. send one request; give up on timeout
  • **At-least-once** semantics
    – Retry on timeout; risk operation occurring again
    – Ok if the operation is read-only, or idempotent
• Note: Assumption of no network duplication
Remote Procedure Call (RPC)

• Request/response protocols are useful – and widely used – but rather clunky to use
  – e.g. need to define the set of requests, including how they are represented in network messages

• A nicer abstraction is Remote Procedure Call (RPC)
  – Programmer simply invokes a procedure...
  – ...but it executes on a remote machine (the server)
  – RPC subsystem handles message formats, sending & receiving, handling timeouts, etc

• Aim is to make distribution (mostly) transparent
  – Certain failure cases wouldn’t happen locally
  – Distributed and local function call performance different
Marshalling arguments

- RPC is integrated with the programming language
  - Some additional magic to specify things are remote
- RPC layer **marshals** parameters to the call, as well as any return value(s), e.g.

```
Caller
  call(…)
  1) Marshal args
  2) Generate ID
  3) Send message
  4) Start timer
  10) Unmarshal return values
  11) Acknowledge

RPC Service
  1) Marshal args
  2) Generate ID
  3) Send message
  4) Start timer
  5) Unmarshal args
  6) Record ID
  7) Marshal return values
  8) Send reply
  9) Set timer

Remote Function
  fun(…)
```
IDLs and stubs

• To marshal, the RPC layer (on both sides!) must know:
  – how many arguments the procedure has,
  – how many results are expected, and
  – the types of all of the above

• The programmer must specify this by describing things in an **interface definition language (IDL)**
  – In higher-level languages, this may already be included as standard (e.g. C#, Java)
  – In others (e.g. C), IDL is part of the middleware

• The RPC layer can then automatically generate **stubs**
  – Small pieces of code at client and server (see previous)
  – May also provide authentication, encryption
  – Provides integrity, confidentiality
Example: SunRPC

• Developed mid 80’s for Sun Unix systems
• Simple request/response protocol:
  – Server registers one or more “programs” (services)
  – Client issues requests to invoke specific procedures within a specific service
• Messages can be sent over any transport protocol (most commonly UDP/IP and later TCP/IP)
  – Requests have a unique transaction id which can be used to detect & handle retransmissions
  – At-least-once semantics
  – Various types of access transparency including byte-order
eXternal Data Representation (XDR)

- SunRPC used **XDR** for describing interfaces:
  
  ```c
  // file: test.x
  program test {
      version testver {
          int get(getargs) = 1;  // procedure number
          int put(putargs) = 2;  // procedure number
      } = 1;                   // version number
      } = 0x12345678;            // program number
  }
  ```

- **rpcgen** generates [un]marshaling code, stubs
  - Single arguments... but recursively convert values
  - Some support for following pointers too
- Data on the wire always in big-endian format (oops!)
Using SunRPC

1. Write XDR, and use rpcgen to generate skeleton code
2. Fill in blanks (i.e. write client/server), compile code
3. Run server & register with portmapper (now: rpcbind)
   – Mappings from { prog#, ver#, proto } -> port
   – (on Linux/UNIX, try “/usr/sbin/rpcinfo -p”)
   – Portmapper is an RPC service on a well-known port
4. Server process will then listen(), awaiting clients
5. When a client starts, client stub calls clnt_create()
   – Sends { prog#, ver#, proto } to portmapper on server, receives port number to use for actual RPC connection
   – Client invokes remote procedures as needed
6. Lately: GSS authentication/encryption (e.g., Kerberos)
Summary + next time

• About this course
• Advantages and challenges of distributed systems
• Types of transparency (+scalability)
• Middleware, the client-server model
• Errors and retry semantics
• RPC, marshalling, SunRPC, and XDR

• Case study: the Network File System (NFS)