Distributed Systems
8L for Part IB

Lecture 1

Dr Robert N. M. Watson

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") which 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 (?)

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

Key changes: loss of global visibility, 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

- Middleware layer extends over multiple machines

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**
# Classic 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>
</tbody>
</table>

Performance or scaling transparency increasingly important

---

# In this Course

- 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 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:

Request-Reply Protocols

- Basic scheme:
  - Client issues a request message
  - Server performs operation, and sends reply
- Simplest version is **synchronous**:
  - client blocks awaiting reply
- Example: HTTP 1.0
  - Client (browser) sends “GET /index.html”
  - Web server fetches file and returns it
  - Browser displays HTML web page
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 was lost  
  2. Request was sent, but server crashed on receipt  
  3. Request was sent & received, and server performed operation (or some of it?), but crashed before replying  
  4. Request was sent & received, and server performed operation correctly, and sent reply ... which was then lost  
  5. As #4, but reply has just been delayed for longer than $T$
- For read-only stateless requests (like HTTP GET), can retry in all cases, but what if request was an order with Amazon?  
  - In case #1, we probably want to re-order... and in case #5 we want to wait for a little bit longer, and otherwise we ... erm?  
- Worse: we don’t know what 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 ;-)
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**
  - 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.
**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
XDR: External Data Representation

- 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 and server moving), compile code
3. Run server program & register with **portmapper**
   - holds mappings from { prog#, ver#, proto } -> port
   - (on Linux/UNIX, try "/usr/sbin/rpcinfo -p")
   - **Portmapper** is itself 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, and gets reply with appropriate port number to use
   - Client now invokes remote procedures as needed
6. Later versions integrated with GSS to provide authentication and encryption – e.g., via Kerberos