Distributed Systems (COMP9243)

Lecture 3: System Architecture

Slide 1
1. System Architectures
   ➜ Client-server (and multi-tier)
   ➜ Peer to peer
   ➜ Hybrid architectures
2. Processes & Server Architecture

Slide 2

Building a Distributed System

Slide 3
Two questions:
1. Where to place the hardware?
2. Where to place the software?

Slide 4
System Architecture:
   ➜ Identifying hardware and software elements
   ➜ Placement of machines
   ➜ Placement of software on machines
   ➜ Communication patterns

Where to place?:
   ➜ Processing capacity, load balancing
   ➜ Communication capacity
   ➜ Locality

Mapping of services to servers:
   ➜ Partitioning
   ➜ Replication
   ➜ Caching
ARCHITECTURE ISSUES
Choosing the right architecture involves:

- Splitting of functionality
- Structuring the application
- Reducing complexity

ARCHITECTURAL PATTERNS

CLIENT-SERVER
Client-Server from another perspective:

Client
Request
Wait for result
Provide service

Server
Request
Reply

How scalable is this?
Example client-server code in C:

```c
client(void) {
    struct sockaddr_in cin;
    char buffer[bufsize];
    int sd;

    // set server address in cin
    sd = socket(AF_INET, SOCK_STREAM, 0);
    connect(sd, (void *)&cin, sizeof(cin));
    send(sd, buffer, strlen(buffer), 0);
    recv(sd, buffer, bufsize, 0);
    close(sd);
}

server(void) {
    struct sockaddr_in cin, sin;
    int sd, sd_client;

    // set server address in sin
    sd = socket(AF_INET, SOCK_STREAM, 0);
    bind(sd, (struct sockaddr *)&sin, sizeof(sin));
    listen(sd, queuesize);
    while (true) {
        sd_client = accept(sd, (struct sockaddr *)&cin, &addrlen);
        recv(sd_client, buffer, sizeof(buffer), 0);
        DoService(buffer);
        send(sd_client, buffer, strlen(buffer), 0);
        close(sd_client);
    }
    close(sd);
}
```

Example client-server code in Erlang:

```erlang
% Client code using the increment server
client (Server) ->
    Server ! {self (), 10},
    receive
        {From, Reply} -> io:format(“Result: ~w”, [Reply])
    end.

% Server loop for increment server
loop () ->
    receive
        {From, Msg} -> From ! {self (), Msg + 1},
            loop ();
    stop -> true
end.

% Initiate the server
start_server() -> spawn (fun () -> loop () end).
```

Splitting Functionality:

Which is the best approach?
Vertical Distribution (Multi-tier)

Three ‘layers’ of functionality:
- User interface
- Processing/Application logic
- Data
  - Logically different components on different machines

Leads to Service-Oriented architectures (e.g. microservices).

Horizontal Distribution

Logically equivalent components replicated on different machines

How scalable is this?

Note: Scaling Up vs Scaling Out?
Horizontal and Vertical Distribution not the same as Horizontal and Vertical Scaling.

Vertical Scaling: Scaling UP Increasing the resources of a single machine

Horizontal Scaling: Scaling OUT Adding more machines.
Horizontal and Vertical Distribution are both examples of this.
**Service Oriented Architecture (SOA)**

- Auction Service
- Stock Service
- Bank Service
- Photo Service
- HTTP
- XML-RPC
- SOAP

**Microware Services**

- ‘Extreme’ vertical distribution
- split application logic into many (reusable) services
- services limited in scope: single-purpose, do one thing really well
- orchestrate execution of services

**Peer to Peer**

- All processes have client and server roles: servant

Why is this special?

**Peer to Peer and Overlay Networks**

- How do peers keep track of all other peers?
  - static structure: you already know
  - dynamic structure: Overlay Network
    1. structured
    2. unstructured

Overlay Network:

- Application-specific network
- Addressing
- Routing
- Specialised features (e.g., encryption, multicast, etc.)
Example:

Unstructured Overlay

(a) Random network  (b) Scale-free network

- Data stored at random nodes
- Partial view: node’s list of neighbours
- Exchange partial views with neighbours to update

What’s a problem with this?

Structured Overlay

Distributed Hash Table:

- Nodes have identifier and range, Data has identifier
- Node is responsible for data that falls in its range
- Search is routed to appropriate node
- Examples: Chord, Pastry, Kademlia

What’s a problem with this?

Hybrid Architectures

Combination of architectures.

Examples:
- Superpeer networks
- Collaborative distributed systems
- Edge-server systems
Superpeer Networks:
- Regular peers are clients of superpeers
- Superpeers are servers for regular peers
- Superpeers are peers among themselves
- Superpeers may maintain large index, or act as brokers
- Example: Skype

What are potential issues?

Collaborative Distributed Systems:
Example: BitTorrent
- Node downloads chunks of file from many other nodes
- Node provides downloaded chunks to other nodes
- Tracker keeps track of active nodes that have chunks of file
- Enforce collaboration by penalising selfish nodes

What problems does BitTorrent face?

Edge-Server Networks:
- Servers placed at the edge of the network
- Servers replicate content
- Mostly used for content and application distribution
- Content Distribution Networks: Akamai, CloudFront, CoralCDN

What are the challenges?

Server Design

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, non-blocking system calls</td>
</tr>
</tbody>
</table>
**Stateful vs Stateless Servers**

Stateful:
- Keeps persistent information about clients
- Improved performance
- Expensive crash recovery
- Must track clients

Stateless:
- Does not keep state of clients
- *soft state* design: limited client state
- Can change own state without informing clients
- No cleanup after crash
- Easy to replicate
- Increased communication

Note: Session state vs. Permanent state

**Request Switching**

Transport layer switch:
- Logically a single TCP connection

DNS-based:
- Round-robin DNS

Application layer switch:
- Analyse requests
- Forward to appropriate server

**Clustering Servers**

Logical switch (possibly multiple)

Client requests

First tier

Second tier

Third tier

**Virtualisation**

What are the benefits?
CONTAINERISATION

What are the benefits?
What are the drawbacks?

[FROM: https://www.docker.com/resources/what-container]

SERVERLESS

Serverless does use servers!
→ You don’t maintain them yourself
→ You only provide functions to run
→ Transparently run on servers
→ Functions as a Service (FaaS)
  - code components have a short lifecycle (per request)
  - environment manages loading, starting, stopping code
  - client-side management of control-flow, application logic

[FROM: https://martinfowler.com/bliki/Serverless.html]

CODE MOBILITY

Why move code?
→ Optimise computation (load balancing)
→ Optimise communication

Weak vs Strong Mobility:
Weak  transfer only code
Strong  transfer code and execution segment

Sender vs Receiver Initiated migration:
Sender  Send program to compute server
Receiver  Download applets

Examples: Java, JavaScript, Virtual Machines, Mobile Agents

What are the challenges of code mobility?
Homework

**Client Server:**

→ Do Exercise *Client server exercise (Erlang)* Part A.

**Hacker’s Edition: Client-Server vs Ring:**

→ Do Exercise *Client-Server vs. Ring (Erlang)*