Building a Distributed System

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

System Architecture:
- placement of machines
- placement of software on machines

Where to place?:
- processing capacity, load balancing
- communication capacity
- locality

Mapping of services to servers:
- Partitioning
- Replication
- Caching
Client-Server from another perspective:

How scalable is this?

Example client-server code in C:

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

    sd = socket(AF_INET, SOCK_STREAM, 0);
    bind(sd, (struct sockaddr *)&cin, 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:

```
% Client code using the increment server
client (Server) ->
    Server ! {self (), 10},
    receive
    {From, Reply} -> io:format("Result: ~w~n", [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:

(a) (b) (c) (d) (e)

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).
**Slide 13**

Vertical Distribution from another perspective:

- **User interface (presentation)**
- **Application server**
- **Database server**

- Request operation
- Wait for result
- Wait for data
- Return data
- Return result
- Time

How scalable is this?

**Slide 14**

**Horizontal Distribution**

- Front end handling incoming requests
- Requests handled in round-robin fashion
- Replicated Web servers each containing the same Web pages
- Disks

- Internet

- Logically equivalent components replicated on different machines

How scalable is this?

**Slide 15**

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.

**Slide 16**

**Peer to Peer**

- All processes have client and server roles: servent, master

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

---

**Slide 21**

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

---

**Slide 22**

What are potential issues?

---

**Slide 23**

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

---

**Slide 24**

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 problems does BitTorrent face?**

---

**What are the challenges?**
### Server Design

#### Dispatcher thread
- Request coming in from the network
- Request dispatched to a worker thread

#### Worker thread
- Operating system

### Model vs. Characteristics

<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

### Clustered Servers

**Logical switch (possibly multiple):**
- Application/compute servers
- Distributed file/database system

### Request Switching

#### Transport layer switch:
- Logically a single TCP connection
- Responses

#### DNS-based:
- Round-robin DNS

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

Virtual Machines

- Host OS
- Virtual Machine Monitor
- Guest OS
- Server

What are the benefits?

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?

Why Communication?

Cooperating processes need to communicate.
- For synchronisation and control
- To share data
In a Non-Distributed System:

Two approaches to communication:

- **Shared memory**

**Slide 34**

**Shared Memory:**

- $x = 12$
- $i = 5$

**Slide 36**

**Message Passing:**

- Process A
- Process B
- Address space 1
- Address space 2

**Slide 35**

**Message passing**
In a Non-Distributed System:

Two approaches to communication:

- **Shared memory**
  - Direct memory access (Threads)
  - Mapped memory (Processes)
- **Message passing**
  - OS’s IPC mechanisms

Communication in a Distributed System

Previous slides assumed a uniprocessor or a multiprocessor. In a distributed system (multicomputer) things change:

**Shared Memory:**
- There is no way to physically share memory

**Message Passing:**
- Over the network
- Introduces latencies
- Introduces higher chances of failure
- Heterogeneity introduces possible incompatibilities

Message Passing

**Basics:**
- `send()`
- `receive()`

**Variations:**
- Connection oriented vs Connectionless
- Point-to-point vs Group
- Synchronous vs Asynchronous
- Buffered vs Unbuffered
- Reliable vs Unreliable
- Message ordering guarantees

**Data Representation:**
- Marshalling
- Endianness

Coupling

Dependency between sender and receiver

**Temporal** do sender and receiver have to be active at the same time?

**Spatial** do sender and receiver have to know about each other? explicitly address each other?

**Semantic** do sender and receiver have to share knowledge of content syntax and semantics?

**Platform** do sender and receiver have to use the same platform?

Tight vs Loose coupling: yes vs no
COMMUNICATION MODES

Data-Oriented vs Control-Oriented Communication:

Data-oriented communication
- Facilitates data exchange between threads
- Shared address space, shared memory & message passing

Control-oriented communication
- Associates a transfer of control with communication
- Active messages, remote procedure call (RPC) & remote method invocation (RMI)

Synchronous vs Asynchronous Communication:

Synchronous
- Sender blocks until message received
  - Often sender blocked until message is processed and a reply received
- Sender and receiver must be active at the same time
- Receiver waits for requests, processes them (ASAP), and returns reply
- Client-Server generally uses synchronous communication

Asynchronous
- Sender continues execution after sending message (does not block waiting for reply)
- Message may be queued if receiver not active
- Message may be processed later at receiver’s convenience

When is Synchronous suitable? Asynchronous?

Transient vs Persistent Communication:

Transient
- Message discarded if cannot be delivered to receiver immediately
- Example: HTTP request

Persistent
- Message stored (somewhere) until receiver can accept it
- Example: email

Coupling? Time

Provider-Initiated vs Consumer-Initiated Communication:

Provider-Initiated
- Message sent when data is available
- Example: notifications

Consumer-Initiated
- Request sent for data
- Example: HTTP request
Direct-Addressing vs Indirect-Addressing Communication:

**Direct-Addressing**
- Message sent directly to receiver
- Example: HTTP request

**Indirect-Addressing**
- Message not sent to a particular receiver
- Example: broadcast, publish/subscribe

**Coupling? Space**

**Combinations:**

- **Persistent Asynchronous**
  - Message can be sent only if B is running

- **Transient Asynchronous**
  - Accepted

- **Persistent Synchronous**
  - Starts processing request

- **Transient Synchronous**
  - Request Received
  - Accepted
  - Request Received
  - Accepted

**Examples?**

**Communication Abstractions**
Abstractions above simple message passing make communication easier for the programmer.
Provided by higher level APIs

1. Message-Oriented Communication
2. Request-Reply, Remote Procedure Call (RPC) & Remote Method Invocation (RMI)
3. Group Communication
4. Event-based Communication
5. Shared Space

**Message-Oriented Communication**
Communication models based on message passing

Traditional `send()/receive()` provides:
- Asynchronous and Synchronous communication
- Transient communication

What more does it provide than `send()/receive()`?
- Persistent communication (Message queues)
- Hides implementation details
- Marshalling
**Example: Message Passing Interface (MPI)**

- Designed for parallel applications
- Makes use of available underlying network
- Tailored to transient communication
- No persistent communication
- Primitives for all forms of transient communication
- Group communication

MPI is BIG. Standard reference has over 100 functions and is over 350 pages long!

**Example: Message Queuing Systems**

- Provides:
  - Persistent communication
  - Message Queues: store/forward
  - Transfer of messages between queues

- Model:
  - Application-specific queues
  - Messages addressed to specific queues
  - Only guarantee delivery to queue. Not when.
  - Message transfer can be in the order of minutes

- Examples:
  - IBM MQSeries, Java Message Service, Amazon SQS, Advanced Message Queuing Protocol, MQTT, STOMP

Very similar to email but more general purpose (i.e., enables communication between applications and not just people)

**Request-Reply Communication**

- Request:
  - a service
  - data

- Reply:
  - result of executing service
  - data

- Requirement:
  - Message formatting
  - Protocol
Example: Remote Procedure Call (RPC)

Idea: Replace I/O oriented message passing model by execution of a procedure call on a remote node [BN84]:
- Synchronous - based on blocking messages
- Message-passing details hidden from application
- Procedure call parameters used to transmit data
- Client calls local "stub" which does messaging and marshalling

Confusing local and remote operations can be dangerous, why?

This is what it's like in RPC:

% Client code
client (Server) ->
  register(server, Server),
  Result = inc (10),
  io:format ({"Result: "w"n", [Result]}).

% Server code
inc (Value) -> Value + 1.

Where is the communication?
RPC Implementation:
1. Client calls client stub (normal procedure call)
2. Client stub packs parameters into message data structure
3. Client stub performs `send()` syscall and blocks
4. Kernel transfers message to remote kernel
5. Remote kernel delivers to server stub, blocked in `receive()`
6. Server stub unpacks message, calls server (normal proc call)
7. Server returns to stub, which packs result into message
8. Server stub performs `send()` syscall
9. Kernel delivers to client stub, which unpacks and returns

---

Example client stub in Erlang:
```erlang
% Client code using RPC stub
client (Server) ->
    register(server, Server),
    Result = inc (10),
    io:format ("Result: ~w~n", [Result]).
```

---

Example server stub in Erlang:
```erlang
% increment implementation
inc (Value) -> Value + 1.

% RPC Server dispatch loop
server () ->
    receive
        {From, inc, Value} ->
            From ! {self(), inc, inc(Value)}
    end,
    server().
```

---

Parameter marshalling:
- stub must pack ("marshal") parameters into message structure
- message data must be pointer free
- (by-reference data must be passed by-value)
- may have to perform other conversions:
  - byte order (big endian vs little endian)
  - floating point format
  - dealing with pointers
  - convert everything to standard ("network") format, or
  - message indicates format, receiver converts if necessary
- stubs may be generated automatically from interface specs
Examples of RPC frameworks:

- **SUN RPC (aka ONC RPC):** Internet RFC1050 (V1), RFC1831 (V2)
  - Based on XDR data representation (RFC1014)(RFC1832)
  - Basis of standard distributed services, such as NFS and NIS
- **Distributed Computing Environment (DCE) RPC**
- **XML (data representation) and HTTP (transport)**
  - Text-based data stream is easier to debug
  - HTTP simplifies integration with web servers and works through firewalls
  - For example, XML-RPC (lightweight) and SOAP (more powerful, but often unnecessarily complex)
- **Many More:** Facebook Thrift, Google Protocol Buffers RPC, Microsoft .NET

---

Sun RPC Example:

**Run example code from website**

---

Sun RPC - interface definition:

```c
program DATE_PROG {
    version DATE_VERS {
        long BIN_DATE(void) = 1; /* proc num = 1 */
        string STR_DATE(long) = 2; /* proc num = 2 */
    } = 1; /* version = 1 */
    } = 0x31234567; /* prog num */
```

---

Sun RPC - client code:

```c
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
...
main(int argc, char **argv) {
    CLIENT *cl; /* RPC handle */
    ...
    cl = clnt_create(argv[1], DATE_PROG, DATE_VERS, "udp");
    lresult = bin_date_1(NULL, cl);
    printf("time on host %s = %ld\n", server, *lresult);
    sresult = str_date_1(lresult, cl);
    printf("time on host %s = %s", server, *sresult);
    clnt_destroy(cl); /* done with the handle */
}
```
Sun RPC - server code:

```c
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
long * bin_date_1() {
    static long timeval; /* must be static */
    long time(); /* Unix function */
    timeval = time((long *) 0);
    return(&timeval);
}
char ** str_date_1(long *bintime) {
    static char *ptr; /* must be static */
    char *ctime(); /* Unix function */
    ptr = ctime(bintime); /* convert to local time */
    return(&ptr); /* return the address of pointer */
}
```

**ONE-way (ASYNCHRONOUS) RPC**

- When no reply is required
- When reply isn't needed immediately (2 asynchronous RPCs - deferred synchronous RPC)

**REMOTE METHOD INVOCATION (RMI)**

Like RPC, but transition from the server metaphor to the object metaphor.

**Why is this important?**
- RPC: explicit handling of host identification to determine the destination
- RMI: addressed to a particular object
- Objects are first-class citizens
- Can pass object references as parameters
- More natural resource management and error handling
- But still, only a small evolutionary step

**TRANSPARENCY CAN BE DANGEROUS**

Why is the transparency provided by RPC and RMI dangerous?
- Remote operations can fail in different ways
- Remote operations can have arbitrary latency
- Remote operations have a different memory access model
- Remote operations can involve concurrency in subtle ways

What happens if this is ignored?
- Unreliable services and applications
- Limited scalability
- Bad performance

See "A note on distributed computing" (Waldo et al. 94)
What are the difficulties with group communication?

Two kinds of group communication:
- **Broadcast** (message sent to everyone)
- **Multicast** (message sent to specific group)

Used for:
- Replication of services
- Replication of data
- Service discovery
- Event notification

Issues:
- Reliability
- Ordering

Example:
- IP multicast
- Roaming

---

**Example: Gossip-Based Communication**

Technique that relies on epidemic behaviour, e.g. spreading diseases among people.

Variant: *rumour spreading*, or *gossiping*.

- When node $P$ receives data item $x$, it tries to push it to arbitrary node $Q$.
- If $x$ is new to $Q$, then $P$ keeps on spreading $x$ to other nodes.
- If node $Q$ already has $x$, $P$ stops spreading $x$ with certain probability.

Analogy from real life: Spreading rumours among people.

---

**Event-Based Communication**

- Communication through propagation of events
- Generally associated with *publish/subscribe* systems
- Sender process publishes events
- Receiver process subscribes to events and receives only the ones it is interested in.
- Loose coupling: space, time
- Example: OMG Data Distribution Service (DDS), JMS, Tibco

---

**Shared Space Communication**
**Slide 73**

**Shared Space Communication**

Example: Distributed Shared Memory:

![Shared global address space diagram](image)

**Coupling?**

**Slide 74**

Example: Tuple Space:

![Tuple instance diagram](image)

**Coupling?**

**Slide 75**

**Reading List**

**Implementing Remote Procedure Calls** A classic paper about the design and implementation of one of the first RPC systems.

**Slide 76**

**Homework**

**RPC:**
- Do Exercise Client server exercise (Erlang) Part B

**Synchronous vs Asynchronous:**
- Explain how you can implement synchronous communication using only asynchronous communication primitives.
- How about the opposite?

**Hacker’s Edition: Client-Server vs Ring:**
- Do Exercise Client-Server vs. Ring (Erlang)