Distributed Systems

Many applications, particularly those with multiple users are most naturally structured as a collection of processes distributed over a number of machines.

Examples include multi-user games and simulations, distance learning tools, and software development environments.

Essence: Stateful interaction at multiple locations.

Kinds of Distributed System

Computations are distributed for different purposes, i.e.

**Computational Systems** Connect multiple clusters or High Performance Computers (HPCs), e.g. Battlefield simulation.

**High Throughput Systems** spread computations across networks of processors, possibly to use idle cycles/hours, e.g. SETI at home.

**Access to Shared Resources** e.g. printer or particle accelerator.

**Distributed Databases** Shared access to data resources, e.g. repositories (databases), or data source like a satellite downlink.

Distributed Functionality

The hardware & software redundancy in enables a distributed system to offer functionality not possible on a single machine:

- **Reliability**: the system functions correctly even in the presence of hardware and software failures, e.g. detect faults, raise an exception, and handle gracefully.
- **Availability**: (most of) the system is almost always functioning, e.g. telephone products aim for 99.999% availability.
- **Scalability**: The throughput of the system can be increased by increasing hardware, e.g. if a 4 processor telephone exchange handles 100 calls/minute, how many can an 8 processor exchange handle?
Distributed System Concepts

**Location.** A physical device that performs computation, typically a processor with memory and associated physical resources such as disk, screen, etc.

- A language is **location independent** if locations are implicit, e.g. access a file regardless of its location.
- A language is **location aware** if locations are explicit, e.g. forking a new thread onto a named location.

**Thread.** An independent sequence of executing instructions. Sometimes also known as a lightweight process to indicate that a thread has minimal private resources.

**Process.** An operating system process with a private address and other resources, like I/O capabilities, and usually threads. A location may reside on a PE or a group of PEs.

**Communication:** the exchange of data & possibly computations.

**Synchronisation:** the coordination of control. Communication and Synchronisation are closely related, e.g. communication requires synchronisation to safely pass data to another thread, and some form of communication is necessary to indicate that synchronisation has occurred.

**Closed Systems** comprise a fixed set of programs executing on a fixed set of locations.

**Open Systems** comprise a dynamic programs set of programs and locations interacting using a predefined protocol, e.g. a client-server model.

---

**Scale of Distribution**

**Large-scale** distributed applications may have components written in multiple languages, supplied by several vendors and executed on a heterogeneous collection of platforms. They also have elaborate mechanisms for managing the failure of components of the system.

Large-scale distribution is supported by standard interfaces like Globus, CORBA or Microsoft DCOM.

**Small-scale** distribution entails components written in a single language, is typically constructed by a single vendor, and is often restricted to an homogeneous network of machines, with a simple model of failures.

Small-scale distributed applications are typically constructed in a distributed programming language, e.g. Java with Remote Method Invocation (RMI). Distributed languages allow the system to be developed in a single, homogeneous, framework, and make the distribution more **transparent**.

We will study

- some small-scale distribution technologies: Sockets, Java/RMI, ...
- some large-scale distribution technologies: Globus, Hadoop, ...
**Common Distributed Paradigms**

**Client/Server** Using an agreed API, client programs issue requests to be performed on the server and receive responses.

```
+----------+  +----------+
| Client 1 | ... | Client N |
+----------+  +----------+
```

```
\  ^  /  \
| \  \  /  |
|  v  v  |
| +----------------+
|   \           |
|   /            |
|   +-----------+
|   | Server    |
|   +-----------+
```

**Exercise:** What are the requests and responses for a print server, a database server and a web server?

---

**Request Broker** Using an agreed API, client programs issue requests to a request broker, that forwards them to one of several servers. Various communication architectures possible.

```
+----------+  +----------+
| Client 1 | ... | Client N |
+----------+  +----------+
```

```
\  ^  /  \
| \  \  /  |
|  v  v  |
| +----------------+
|   \  | Request Broker |
|   /  |
|   +---+
|   | /Reqs
|   v
| +----------+  +----------+
| Server 1 | ... | Server M |
+----------+  +----------+
```

---

**Publish/Subscribe** allows greater flexibility than the client/server or request brokers because the publishers are not programmed to send their messages to specific receivers, i.e. subscribers or clients. Rather, published messages are characterized into classes and subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what (if any) publishers there are.

Subscribers may also republish messages. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology. The publish/subscribe pattern is closely related to event-driven programming and the observer design pattern.

Example publish/subscribe systems include RGMA, Conoise-G
Sockets
Sockets are the 1st small-scale distribution paradigm we study.

Socket API or sockets is now
- de facto industry standard API for TCP/IP
- natively supported by OS under many UNIXes
- supported by library on other OSs like Windows95, NT, Solaris

A Socket
- is a communication endpoint with associated protocol
- is used for inter-process communication within communication domain
- exists for as long as process holds descriptor of it
- supports variety of styles of communication

Types of Socket
Stream socket models byte stream communication that
- is connection-oriented
- is reliable, order preserving and flow controlled
- supports out of band communication

Datagram socket models byte stream communication that is
- connectionless
- unreliable
- where each communication packet
  - must be addressed
  - may be duplicated or lost in transit

Socket addresses
- are immutable once given
- are unique in address domain
- belong either to Unix or Internet address domain

Sockets are created in Java:
- DatagramSocket(port) constructor
- MulticastSocket(port) constructor
- ServerSocket(port) constructor
- Socket(host, port) constructor

Java Socket Transmission
Depends on socket type,
- datagram: send() and receive() handle packets
- stream: read() and write() operate on I/O streams

TCP protocol implementing stream sockets
- ensures data is not lost or duplicated
- lets sockets be kept warm with dummy transmission every minute
- breaks link if feasible transmission not done in reasonable time
Stream Sockets in Java

Client.java sends a host a message on port 6001

```java
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String argv[]) {
        try {
            Socket socket = new Socket(argv[0], 6001);
            OutputStream output = socket.getOutputStream();
            output.write(argv[1].getBytes);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Client.java is run with 2 arguments - a host and a message:

```
host% java Client linux23 "Spring-time for Hitler"
```

Server.java receives and prints out message sent on port 6001.

```java
import java.io.*;
import java.net.*;

public class Server {
    public static void main(String args[]) {
        byte[] buf = new byte[1024];
        try {
            ServerSocket s = new ServerSocket(6001, 5);
            Socket sock = s.accept();
            InputStream input = sock.getInputStream();
            int n = input.read(buf);
            System.out.println("Message: " + new String(buf, 0, n));
            sock.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Server.java is run on linux23:

```
linux23% java Server
Message: Spring-time for Hitler
```

Socket Classification

Sockets

- are location aware (socket addresses name location)
- can be used to build closed or open systems
- used for small scale distribution as so low level
Critique of Sockets

Sockets are

- simple and well understood
- generic way of communicating via byte streams or datagrams
- widely supported by programming languages and OSs

Sockets lack

- architecture neutral encoding of data
- programming language data types
- application level protocol for communication
- handling of exceptional conditions
- more sophisticated styles of communicating

Socket communication is low level and forces programmers to design & implement ad hoc means to

- communicate data types in architecture neutral way
- code and decode program data types for communication
- serialise transmission of aggregate and bulk data types
- handle exceptions like unresponsive communication peer
- use communication intermediaries - proxies, brokers etc.

A Generic form of inter-process communication that handled these issues would be useful.