Distributed System Introduction

Advantages/Reasons

- **Communication**: allows different components communicating with each other, multiple users can communicate with each other concurrently, e.g. zoom meetings
- **Remote access**: user can access components remotely, e.g. control machine on Mars
- **Resource sharing**: single resource can be shared among multiple users: significant cost saving and increased utilization
- **Reliability**: increased mean-time-to-failure, redundancy in resources allows the system to continue to operate in the presence of hardware and software failures
- **Availability**: less downtime, e.g. when upgrading some hardware the system can stay online on remaining hardware
- **Scalability**: using more resources to provide greater capacity than any single resource can ideally having N resources provides N times the capacity of a single resource
- **Function separation**: we can have client and servers, we can have seperate servers for different functions to achieve low coupling, e.g. one server for data collection communicating with another server for data processing
- **Inherent distribution**: we can have servers all around the world to serve people locates near the server with a shorter waiting time

Cost

- **Communication complexity.** It takes addition time and power consumption

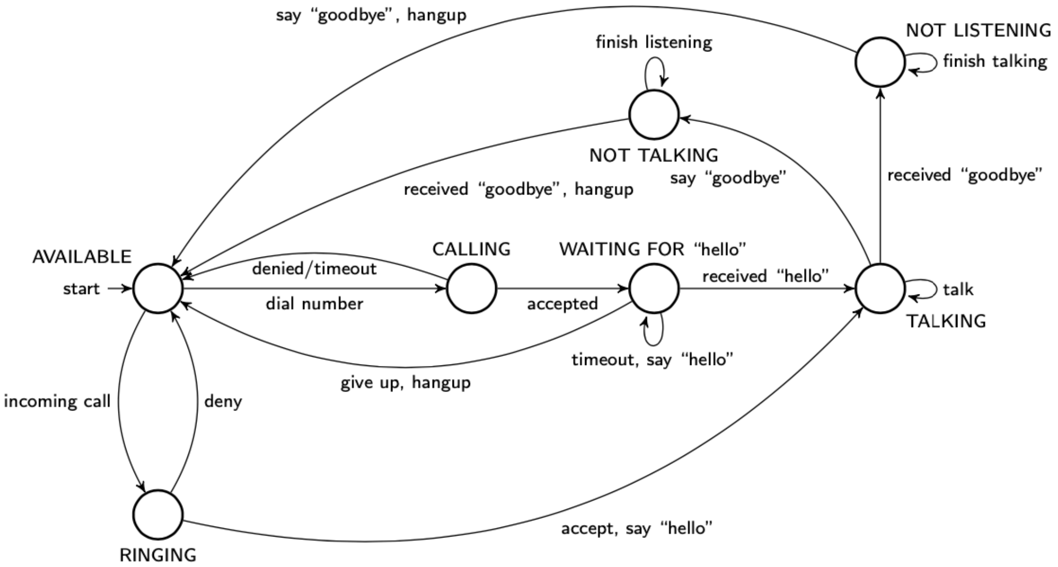
Design Challenges

- **Heterogeneity** – the parts of the system are not consistent (homogenous), such as hardware, OS, networking, processes written languages can differ in components.
  Solution: use middleware
- **Openness** – the system can be built upon or accessed by third-party developers through public APIs and protocols. APIs should be stable, simple to use and provide all the required features.
  Solution: Reasonable interface
- **Security** - confidentiality, integrity and availability. Safety of communication, stability under attack, and safety of execution of third-party code
  Solution: encrypting, authorisation, authentication, stability of service (anti-attack)
- **Scalability** - increasing the number of system components increases the overheads. For a distributed system operation to be scalable, we would like the operation's overheads to grow no faster than log N.
  Solution: reasonable interfaces and architecture
- **Failure Handling** - system components fail independently, and the communication network can also fail. It is hard to know if a component has failed or is just taking longer to respond.
  Solution: see chapter failure handling in Models
- **Concurrency** - Autonomous components execute concurrent tasks and multiple processes, leading to a higher level of concurrency control requirements.
  Solution: file locks
- **Transparency** - The users may find it troubling to learn how to use the system, and attackers may attack according to the system stage. We need to hide aspects of the system from the high layers of application.
  - access, location, concurrency, replication, failure, mobility, performance, scaling transparency
  - Solution: never show the server stage to the user
- **No global clock.** Distributed systems have to deal with the fact of not having a global clock (time) and implement time synchronization techniques when needed.
  Solution: Clock Drift Rate
- **No global state**: No single process can have knowledge of the current global state of the system
  Solution: Middleware/monitoring
- **Communication** via message passing, No shared memory.
  Solution: RPC/RMI, Socket, HTTP, etc
- **Resource sharing**: Printer, database, other services -
  Solution: locks

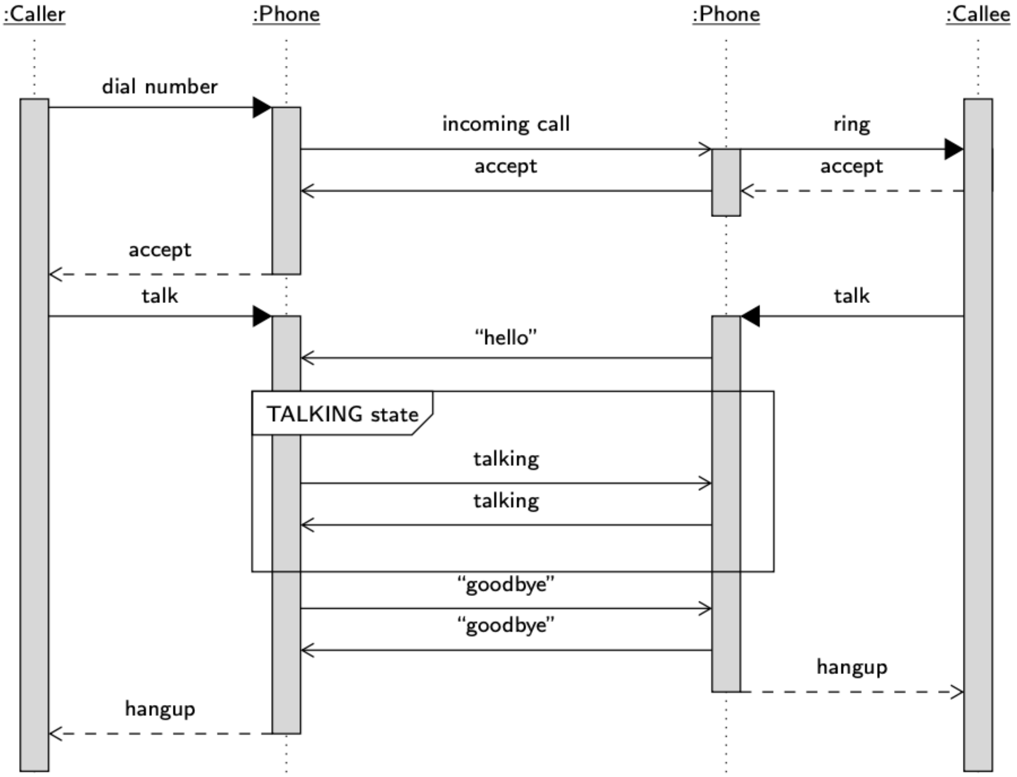
Interprocess communication (IPC)

IPC is the exchange of data between processes.

Example Communication Diagram



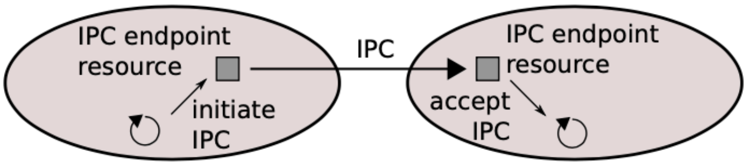
Example Interaction Diagram



Client / Server roles

A process, acting as a *client*, initiates IPC to an endpoint of the server.

A process, acting as a *server*, creates one or more IPC endpoints and waits for a client to initiate communication via one of these endpoints.



- A process may act as both a client and a server, allowing processes to initiate IPC to it, and also initiating IPC to other processes

TCP coding

Server example

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

public class TCPServer {
    public static void main(String[] args) {
        int port = Integer.parseInt(args[0]);
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            while (true) {
                Socket socket = serverSocket.accept();
                new Connection(socket) {
                    @Override
                    public void run() {
                        if (expectMsg("hello")) writeMsg("ACK");
                        else {
                            writeMsg("NAK");
                            try {
                                socket.close();
                            } catch (IOException e) {
                                throw new RuntimeException(e);
                            }
                        }

                        // close the socket when all work done
                        try {
                            socket.close();
                        } catch (IOException e) {
                            throw new RuntimeException(e);
                        }
                    }
                };
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Client example

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;

public class TCPClient {
    public static void main(String[] args) throws UnknownHostException {
        InetAddress ip = InetAddress.getByName(args[0]);
        int port = Integer.parseInt(args[1]);
        try {
            Socket socket = new Socket(ip, port);
            new Connection(socket) {
                @Override
                public void run() {
                    writeMsg("hello");
                    if (!expectMsg("ACK")) {
                        try {
                            socket.close();
                        } catch (IOException e) {
                            throw new RuntimeException(e);
                        }
                    }

                    // close the socket when all work done
                    try {
                        socket.close();
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
            };
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Connection Thread example

```
import java.io.*;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public abstract class Connection extends Thread {

    Socket socket;
    public BufferedReader bufferedReader;
    public BufferedWriter bufferedWriter;

    public Connection(Socket socket) {
        try {
            // set up reader / writer
            this.socket = socket;
            InputStream inputStream = socket.getInputStream();
            OutputStream outputStream = socket.getOutputStream();
            bufferedReader = new BufferedReader(new InputStreamReader(inputStream,
StandardCharsets.UTF_8));
            bufferedWriter = new BufferedWriter(new OutputStreamWriter(outputStream,
StandardCharsets.UTF_8));
            this.start();
        } catch (IOException e) {
            System.out.println("ClientConnection:" + e.getMessage());
        }
    }

    public void writeMsg(String s) {
        try {
            bufferedWriter.write(s);
            bufferedWriter.newLine();
            bufferedWriter.flush();
            System.out.println("write: " + s);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public String receiveMsg() {
        try {
            String s = bufferedReader.readLine();
            System.out.println("receive: " + s);
            //read a message
            return s;
        } catch (IOException e) {
            System.out.println("ClientConnection:" + e.getMessage());
            return null;
        }
    }

    public boolean expectMsg(String s) {
        return receiveMsg().equals(s);
    }

    public abstract void run();
}
```

UDP Coding

Client core

```
DatagramSocket socket = new DatagramSocket();
// send a message
String message = "hello";
byte[] m = message.getBytes();
InetAddress serverAddress = InetAddress.getByName("localhost");
Int serverPort = 8888;
DatagramPacket request = new DatagramPacket(m, message.length(), serverAddress, serverPort);
socket.send(request);
//receive message
byte[] buffer = new byte[1000];
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
socket.receive(reply);
String s = new String(reply.getData());
```

Server core

```
DatagramSocket aSocket = new DatagramSocket(6789); // create socket at agreed port
byte[] buffer = new byte[1000]; // magic number :-S
while(true){
   DatagramPacket request = new DatagramPacket(buffer, buffer.length);
   System.out.println("Server waiting to receive data");
   aSocket.receive(request);
   System.out.println("Received Data: " + new String(request.getData()));
   DatagramPacket reply = new DatagramPacket(request.getData(),
         request.getLength(), request.getAddress(), request.getPort()); aSocket.send(reply);
}
```

Models

- Functional models: how the distributed system implements functionality, the tangible aspects of the distributed system.
- Non-functional models: aspects which are intangible such as time, reliability and security.
- Architectural Model: concerned with the placement of components and entities and relationship between them. Examples:
  - Client-Server and peer process models
  - Client-Server can be modified by
    - The partitioning of data/replication at cooperative servers
    - The caching of data by proxy servers or clients
    - The use of mobile code and mobile agents
    - The requirements to add or remove mobile devices.

Process roles

User Interface Process - Allow interaction between end user and distributed system
Data cache - allow other process get frequent used data faster
Communication relay - allow processes communicate with each other
Job manager - start, terminate and monitor processes in distributed system
Index or registry server - provide location of resources
Database server - provide database service
Authentication server - provide security, authenticate user

Mathematical models

Let $i \in N = \{1, 2, \dots, n\}$ is a process, refereed to as process i
Let machine $j \in M = \{1, 2, \dots, m\}$ that process i is placed on
Let L_j be the number of processes that are placed on machine j: $L_j = |\{i \mid P_i = j\}|$
Let k be maximum capacity of each machine.
Let C_j be capacity of machine j. The case when $C_j = K$ for all j is the (special) homogeneous case of the heterogeneous model.
Let $C_{j,resource}$ be the capacity of a specific resource on machine j, e.g. $C_{j,cpu}$
Let $P_{i,r}$ be the amount of resource r that process i requires.

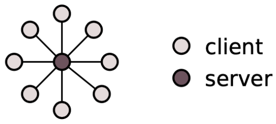
The total resource requirements for resource r on a given machine j is: $L_{j,r} = \sum_{i \mid P_i = j} P_{i,r}$

no machine is allowed to execute more processes than it has capacity for: $L_{j,r} \leq C_{j,r}$ for all $r \in R$

minimizing the maximum consumption of resource r over all machines $\underset{\{P_1,P_2,\dots,P_n\}}{\arg\min} \max_j \{L_{j,r}\}$

minimizing the average percentage resource consumption over all machines $\underset{\{P_1,P_2,\dots,P_n\}}{\arg\min} \max_j \left\{ \frac{1}{|\{\mathcal{R}\}|} \sum_r \frac{L_{j,r}}{C_{j,r}} \right\}$

Centralised Pattern



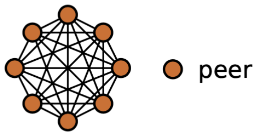
Advantages

- Relatively easy to implement.
- Since clients never directly communicate, they are not exposed to each over the network – privacy/ security.
- If some clients fail or are attacked, it does not have an impact on other clients.

Disadvantages

- Potential data loss: All data is kept at the server.
- If the server fails or is attacked, the distributed system fails – single point of failure.
- A single server may not be able to support a large number of clients – scalability bottleneck.
- Depending on the geographic/network location of the server to the clients, different clients may see different performance characteristics – latency may vary among clients.

Decentralised Pattern



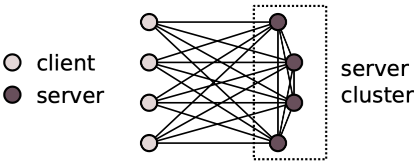
Advantage

- The failure of any peer is no worse than the failure of any other peer – the distributed system is very robust to process failure.
- Resource capacity increases for the system as a whole for each new peer included in the system – scalability.
- Data is distributed evenly

Disadvantage

- Harder to implement than a centralized pattern – processes need to be both clients and servers.
- Data is distributed evenly over the peers.
- Since peers are exposed to each other over the network their IP address becomes public – privacy/ security.

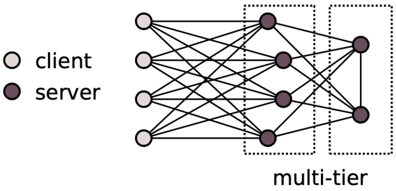
Multi-server Pattern



When a single server no longer supports the number of clients connecting to it, the multi-server pattern can be used to increase overall capacity.

- every server is like a peer to other servers – they can communicate directly with each other, harder to implement than a single server, data is distributed over the servers, the failure of a single server does not lead to system failure.
- The multi-server architecture exposes more server IPs to the clients – potential security issue.
- Servers can be geographically located at places that provide more uniform access for clients.

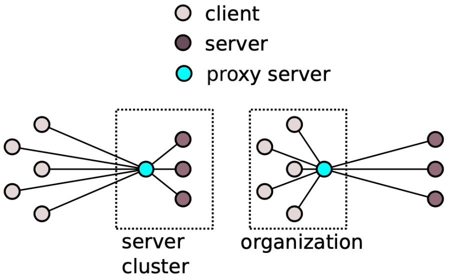
Multi-tier Pattern



The multi-tier pattern is a horizontal view of distributed system functionality (as opposed to a vertical view given by layers).

- The tier furthest from the clients tends to be the data storage tier that maintains all of the data for the distributed system: direct communication with clients never occurs which increases security.
- The tier closest to the clients typically never communicates among itself: this provides isolation and less overheads for synchronization. This tier accesses the data storage tier to obtain data for the clients.
- In a sense, each tier’s processes are like client processes for the next tier inwards.
- More complicated than a multi-server and latency grows with the increase in the number of tiers.

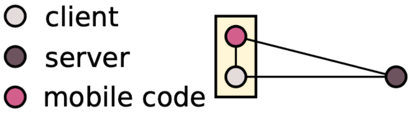
Proxy Pattern



A proxy is a process that sits between a client and a server, relaying request and responses.

- Clients never communicate directly with the server(s) – extra latency.
- All data flows through the proxy – bottleneck and single point of failure.
- A proxy used at the server side can relay requests to appropriate servers, e.g. a web proxy can relay to web servers and websocket servers.
- A proxy used at the organization side can manage client communication to the servers – security.
- Privacy: the proxy can hide the heterogeneity on the other side, more privacy and security

Mobile code and mobile agents



In the mobile code pattern, a process provided by the server is executed on the client's machine, or vice versa. In a mobile agent pattern (not explicitly shown), data is copied along with the mobile code.

- For mobile code, offloading the work to the client reduces load on the server, or in the opposite direction, running client code on the server can avoid load at the client. Both cases avoid some amount of network communication. Mobile code is extremely effective as seen by Javascript in the browser as an example. It leads to "fat clients" (as opposed to thin clients such as the browser without any mobile code). Before Javascript became popular we used Applets as mobile code.
- A mobile agent is mobile code that also maintains state that is copied with the agent's code.
- A mobile agent can move from one server to another, running tasks and collecting data. E.g. if the client requires to undertake a complex query, it may be better to express that as a mobile agent that is executed on the servers that have the data, since undertaking the query on the client may use excessive network resources.
- Executing code supplied by untrusted parties is a high security risk.
- Heterogeneous resources are a challenge.

Adding time to the model

The model so far is static or time-independent. We can allow all of the model parameters to vary with time t, making the model dynamic or time-dependent:

- $n(t)$, $N(t)$, – the number of and ids of processes in the distributed system at time t
- $m(t)$, $M(t)$, – the number of and ids of machines in the distributed system at time t
- $P_i(t)$, $P_{i,r}(t)$ – the location of process $i \in N(t)$ and its requirement for resource r at time t
- $C_j(t)$, $C_{j,r}(t)$ – the resource r capacity for machine $j \in M(t)$ at time time t
- $E_n(t)$, $E_m(t)$ – the IPC that occurs at time t and the machine connections and network characteristics at time t

Failure

Failure which arises from:

- faults – bug in code or defect in hardware
- exceeding specified operating conditions – random environmental conditions, such as mechanical shock, electromagnetic interference, exceeding capacity such as running out of storage space, or user behaviour, that exceeds the range specified for "failure-free" operation, including operation of hardware beyond its specified lifetime.

A Failure Model states the types of failure that are being considered, i.e. what types of failure will the distributed system be subject to, and therefore exclude all remaining types of failure.

Failure manifests in a number of ways, listed here in a loose order of severity:
error indicators: error codes returned from methods including returning null values, cases where a method call has failed, and error messages returned from other processes to indicate some kind of failure
exceptions – may terminate threads if they are not handled, may terminate the process if the main thread terminates, e.g. segment fault
process termination – OS or user terminates the process, e.g. out of memory
OS/hardware failure – the OS and/or machine resources fail
power supply failure – many machines connected to the same power supply may fail at the same time
network failure – computer network equipment can fail as well

Failure Detection:
Silent failure – there is no failure reporting mechanism such as error codes or exception, processes must explicitly detect such failure . e.g. packet loss, program hanging - deadlock, distributed failure - process deadlock, OS failure, device failure
Failure of a remote process in a distributed system is not automatically reported to other processes in the distributed system – it must be detected

- no response to communication requests could be either
- requests “refused” by the remote machine, which is indicative of remote process failure
- timeout - being unresponsive is considered failure

Failure handling
fail-stop – the failure (termination or otherwise) of a process can be detected certainly by other processes in the distributed system
graceful failure – the distributed system can continue to operate, probably with degraded performance, in the presence of faults.
tolerating faults – in this case the users of the system are required to tolerate a certain amount of failure: “can’t provide service, please try again later”. Most users of commercial operating systems tolerate failure in the OS: they restart their computer. They don’t take it back to the store and say it crashed and doesn’t work and they want another one.
failure masking – the distributed system attempts to hide the failure by e.g. retrying the operation transparently.
failure recovery – in some cases, failure needs to be recovered from, because the state of the distributed system has been affected by the failure, e.g. data has been lost, or else the state of the system is no longer valid.

Modelling and analysing failure
Mean time to failure (MTTF) which gives rise to a failure rate: $\lambda = 1 / \text{MTTF}$
The MTTF is a measure of reliability. Higher MTTF means greater reliability.
Probability of the system failing within the next t seconds: $\mathbb{P}[\text{failure} < t] = 1 - e^{-\lambda t}$

The number of failures, x, that occurred in the time interval Δ $\mathbb{P}[\text{failures} = x; \lambda, \Delta] = \dfrac{(\lambda \Delta)^x e^{-\lambda \Delta}}{x!}$

Consider n processes, the probability that none of the processes fail within the next t seconds is:

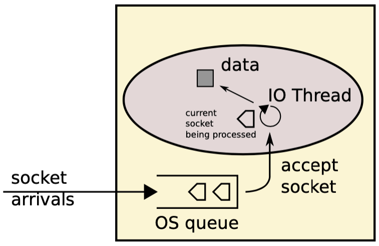
$$(1 - \text{P[failure} < \text{t]})^n = (e^{-\lambda t})^n = e^{-n\lambda t}$$

The probability that at least one of the processes fails within the next t seconds is therefore: $1 - e^{-n\lambda t}$
For example, if MTTF for each process is 1 week, i.e. each process fails independently at randomly on average after 1 week of operation, then for a distributed system with 10 such processes, the probability that at least one of the processes has failed in the first day of operation is: $1 - e^{-10/7} \approx 0.76$
As the number of components in a system increase, the system’s chance of failing increases.

Availability Model
While we are talking in terms of probability, if we let X = 1 if the distributed system is available for use when we try to use it and X = 0 otherwise, and we model X as a random variable, then the availability can be defined as: P[X = 1]
i.e. the probability that the distributed system is available for use when we try to use it. If the availability is 0 then the system is never available, and if the availability is 1 then it is always available. The availability can also be seen as the fraction of total time for which the system is available, e.g. 6.9 days of every 7 days the system is available and 0.1 days it is not. Thus $P[X = 1] \Rightarrow 6.9/7 \approx 0.99$.

Threads

Socket model



- incoming socket connections – communication packets such as used in the TCP protocol, that are requesting that a socket connection be established.
- OS queue – If the socket connection is valid, i.e. to a port that a process is bound to, then the socket connection is put into an OS queue.
- IO thread – If a process binds to a port for socket based communication then it needs to have at least one thread that accepts socket connections and processes them, e.g. accessing local data, reading/writing to the socket.

Queueing theory

- Let λ be the mean rate of request per second that the socket connection requests arrive at
- Let μ be the service rate, the mean number of socket connection requests that the IO thread can process per second.
- Since the queue has finite capacity k – 1 (and a further 1 socket connection possibly being currently processed by the IO thread) it can sometimes become full and new socket requests received by the OS will have to be dropped.
- Let ρ be server utilisation = λ / μ, ρ ≤ 1
- The drop rate is equivalently the blocking probability of the queue:

$$P_{drop} = \begin{cases} \frac{\rho^k(1-\rho)}{1-\rho^{k+1}} & \rho \neq 1 \\ \frac{1}{1+k} & \rho = 1 \end{cases}$$

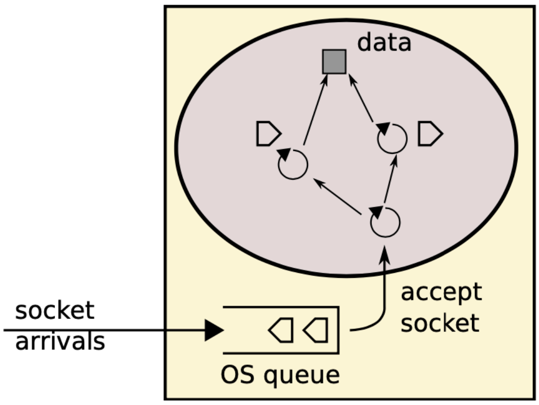
- Effective socket request rate that the IO thread sees: $\lambda_{eff} = \lambda(1 - P_{drop})$.
  e.g. if λ = 10qps, $P_{drop} = 0.1$, then $\lambda_{eff} = 10 * 0.9 = 9$qps
- Let ρ_{eff} be the effective thread utilisation is then λ_{eff} / μ ,
  - If $\lambda_{eff} < \mu$ the system is said to be stable.
  - When $\lambda_{eff} \geq \mu$ then the thread will eventually be busy 100% of the time, meaning it will eventually fall behind, and the system is said to be unstable.
- The average number of socket requests in the system:

$$L = \begin{cases} \frac{\rho}{1-\rho} - \frac{(k+1)\rho^{k+1}}{1-\rho^{k+1}} & \rho \neq 1 \\ \frac{k}{2} & \rho = 1 \end{cases}$$

- If ρ → 1, L will rapidly increase. So we want a bounded queue, since memory consumption can sharply increase under high load – even if the load does not exceed the capacity of the server.

thread-per-connection paradigm

creates a new thread for every socket connection:



Advantages

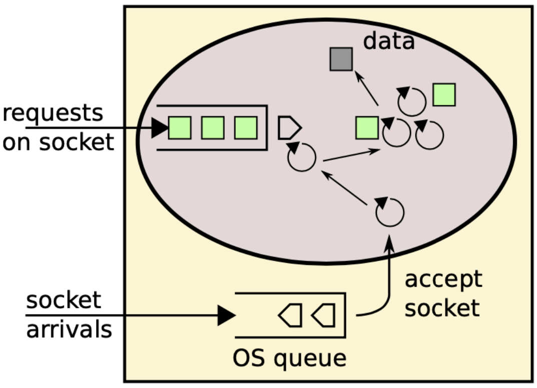
- Easier to implement
- Easier to manage threads
- Number of threads is relatively controllable comparing to thread-per-request paradigm

Disadvantages

- If one single client is sending large number of requests simultaneously, the requests may be stacking
- As the total number of threads going beyond the total number of cores, then the rate of context switching and context state space increases as well. Excessive context state needs to be stored in cache and pushes useful data out of the cache.
- Multiple threads leads to concurrency control requirements, comparing to single threaded programs.

thread-per-request paradigm

allows the thread that is processing the socket to create threads for each request received on the socket:



Advantage over thread-per-connection paradigm

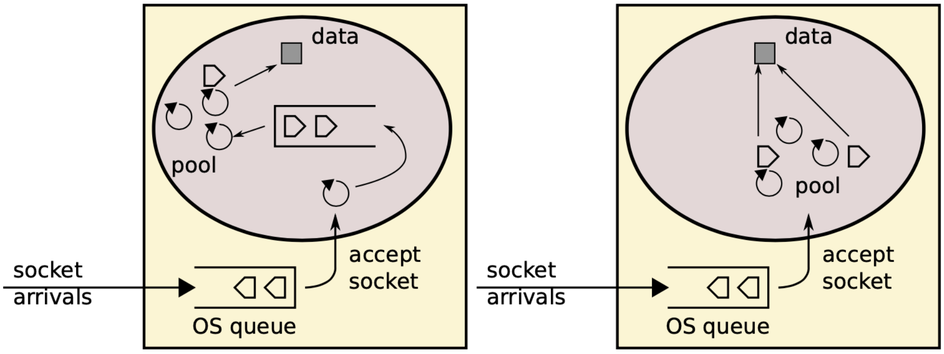
- Thread-per-request can handle one single client sending large number of requests simultaneously problem.

Disadvantage

- Even more threads are potentially created than the thread-per-connection paradigm alone.
- Some protocols allow only a single request per socket connection and so thread-per-request is not useful in this case.
- Requests may have dependencies between them, in that the order that the requests are processed may be important.
- It require more resources, so not suitable in situations with large amount of requests

Thread pool paradigm

creates a fixed or dynamically resizable set of threads that either take incoming socket connection requests from a process maintained queue, or, if the accept socket API is thread safe then the pool of threads can take directly from the OS queue.

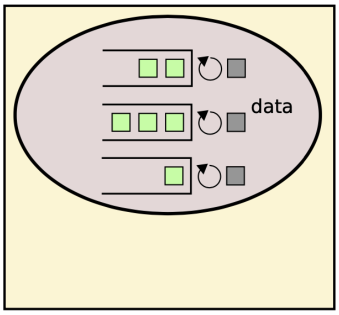


Advantage

More efficiency, avoid time spent on frequent creating and destroying threads
Better scalability, customise based on device capacity

thread-per-object paradigm

creates a thread for each data object.



Advantages

- This can potentially reduce cache miss rates in the machine since for a given data object, only 1 thread ever accesses it and it will reside only in the cache for that thread.
- This can greatly increase cache efficiency which can significantly improve overall performance of the machine.

Disadvantages

- If the usages of object is not distributed evenly the requests may stacking, e.g. object 1 has 100 request per second while object 2 has 1 request per second.

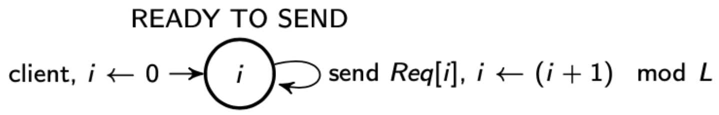
Protocols

Request, Response and Acknowledgement

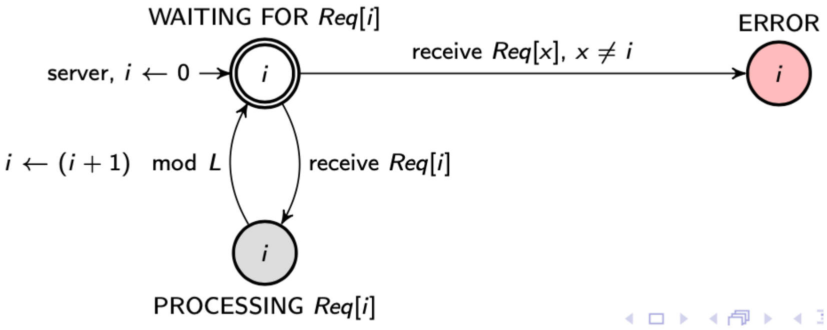
- Request with sequence number i from the client will be written as Req[i].
- Response to Req[i] will be written as Rsp[i].
- Acknowledgement of Rsp[i] will be written as Ack[i].

Request protocol

client's sender protocol



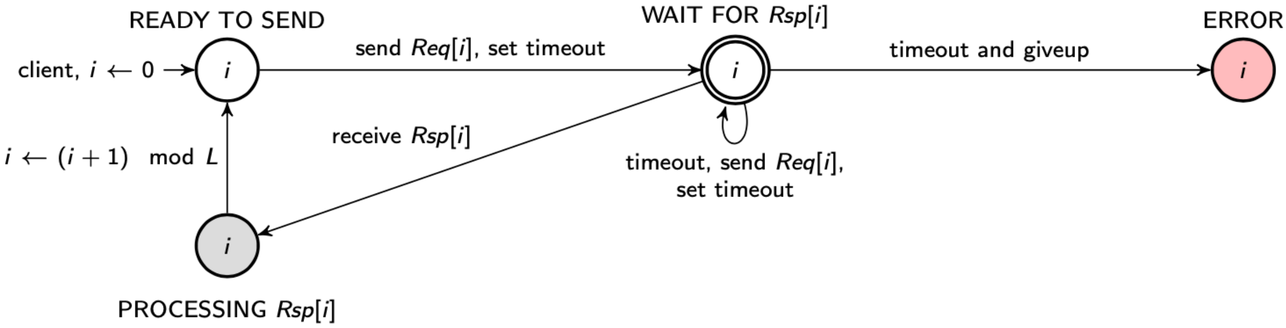
Server's receiver protocol



Disadvantage:
Maybe semantics: client has no information about server,

- whether the request is processed or not,
- client will never arise an error.

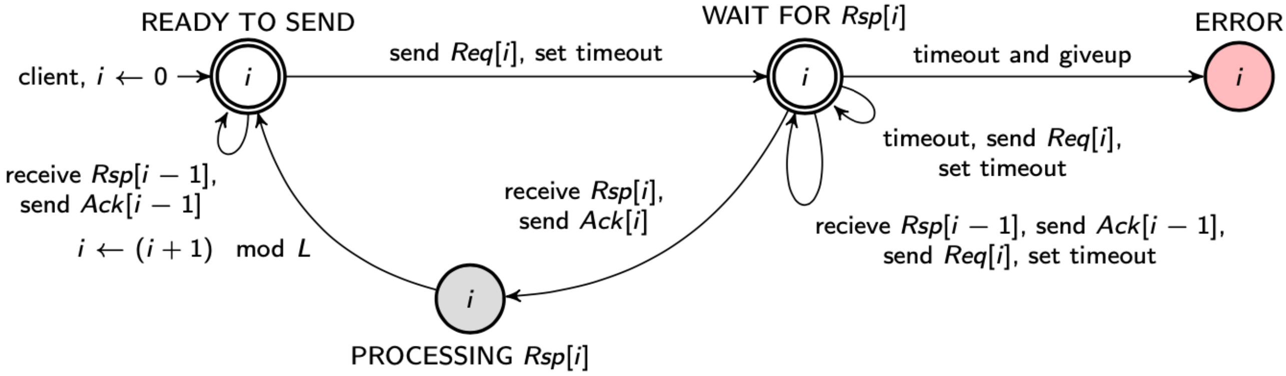
Request/reply protocol



Ensuring sequence is synchronous – it does not allow another request to be sent until it has received a response for the current request. Ensuring that the request has been processed may be impossible. It may eventually give up and the protocol is then in error (exception raised to the client), or it may continue to timeout and retry forever, which blocks the client from sending more requests. We can use at-least semantics or exactly-once semantics. See invocation semantics later.

Request/reply/acknowledge protocol

The client will need to send acknowledgements after receiving a response, perhaps multiple times due to acknowledgements being lost.
Since acknowledgements do not represent any cached data, there is no notion of that at the client.


READY TO SEND
client, $i \leftarrow 0$
i
send $Req[i]$, set timeout
WAIT FOR $Rsp[i]$
i
timeout and giveup
ERROR
i
receive $Rsp[i-1]$, send $Ack[i-1]$
$i \leftarrow (i+1) \mod L$
receive $Rsp[i]$, send $Ack[i]$
timeout, send $Req[i]$, set timeout
recieve $Rsp[i-1]$, send $Ack[i-1]$, send $Req[i]$, set timeout
i
PROCESSING $Rsp[i]$

Invocation Semantics

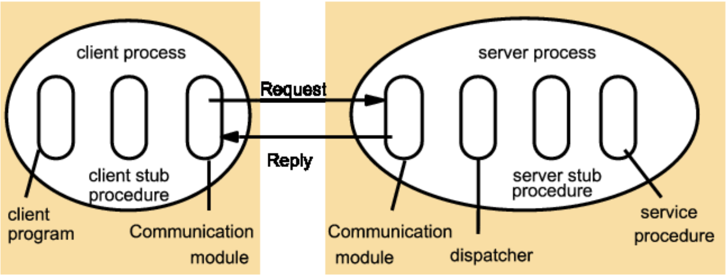
Middleware that implements remote invocation generally provides a certain level of semantics:

- **Maybe**: The remote procedure call may be executed once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called. (Call once, maybe called successfully maybe not)
- **At-least-once**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received an exception to indicate the remote procedure was not executed at all.
  - Call until receive success response, the call 100% happened but unsure happened how many times
  - Server execute all requests and send response
  - Use at least once operations if the request is idempotent, that is, processing it multiple times will not lead to an error, e.g. get operations, updating name etc
- **Exactly-once**: The remote procedure call was either executed exactly once, in which case the caller received a response, or it was not executed at all and the caller receives an exception. (Call and success exactly once)
  - server side cache result of last execution
  - identify whether the request has been resolved once (i.e. Is there a same request received previously?), if so then resend the cached result respond.
- Java RMI supports Exactly-once invocation. Sun RPC supports at-least-once semantics.

Fault tolerance measures			**Call semantics**
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	
No	N/A	N/A	Maybe
Yes	No	Re-execute procedure	At least once
Yes	Yes	Retransmit reply	Exactly once

Remote Procedure Call

RPCs enable clients to execute procedures in server processes based on a defined service interface.

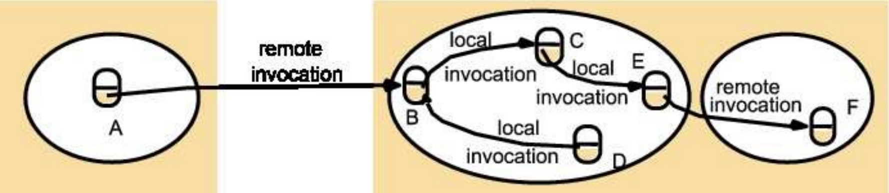


- **Communication Module** Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.
- **Client Stub Procedure** Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module. Unmarshalls the results in the reply.
- **Dispatcher** Selects the server stub based on the procedure identifier and forwards the request to the server stub.
- **Server stub procedure** Unmarshalls the arguments in the request message and forwards it to the service procedure. Marshalls the arguments in the result message and returns it to the client.

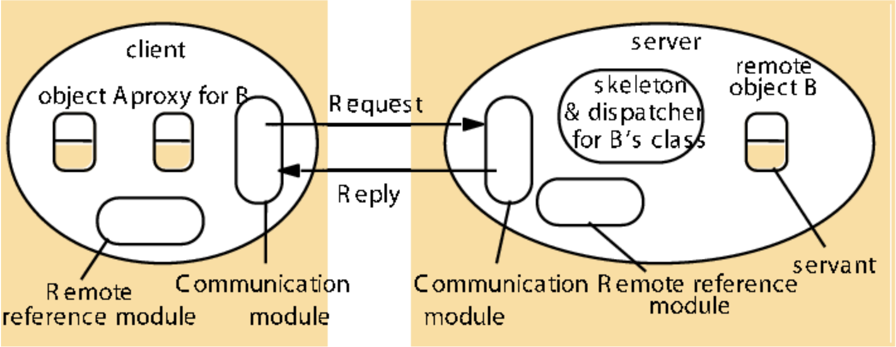
Aim: hides complex network connection and information transmission from user, let users use remote object easier just like using local objects. Developers do not need to care about marshalling, unmarshalling, connection, all these can be done with RPC framework.

Remote Method Invocation

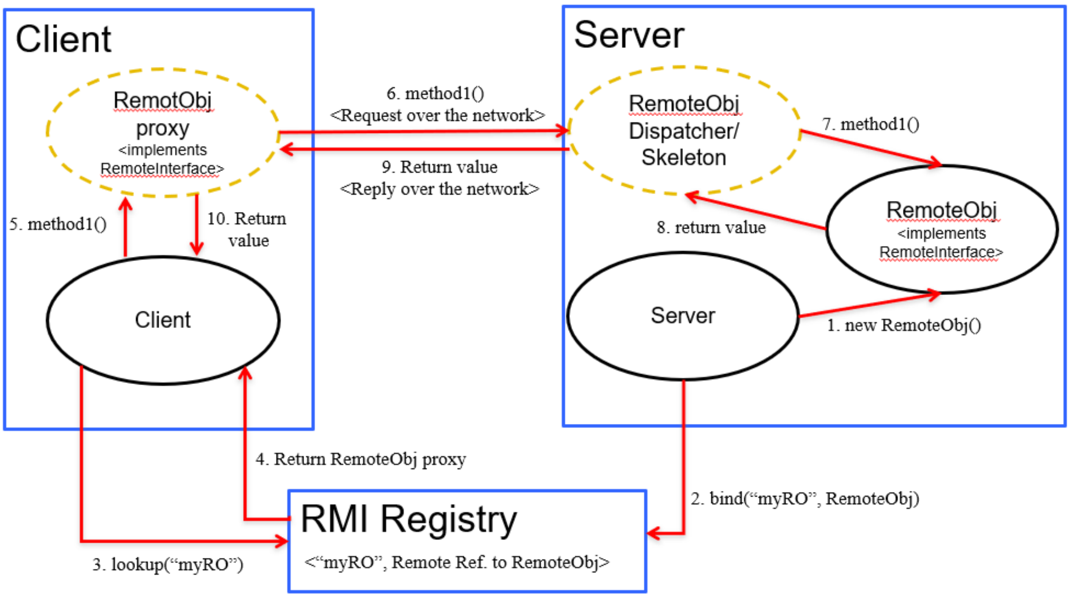
An object that can receive remote invocations is called a remote object. A remote object can receive remote invocations as well as local invocations. Remote objects can invoke methods in local objects as well as other remote objects.



A remote object reference is a unique identifier that can be used throughout the distributed system for identifying an object. This is used for invoking methods in a remote object and can be passed as arguments or returned as results of a remote method invocation.



- The **Communication Module** is responsible for communicating messages (requests and replies) between the client and the server.
- Messages include message type, request ID and remote object reference.
- The **Remote Reference Module** is responsible for creating remote object references and maintaining the remote object table which is used for translating between local and remote object references.
- The **Proxy** plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:
  - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module.
  - Unmarshalling the results and forwarding them to the invoking object
- There is one **Dispatcher** for each remote object class. It is responsible for mapping to an appropriate method in the skeleton based on the method ID.
- The **Skeleton** is responsible for:
  - Unmarshalling the arguments in the request and forwarding them to the servant.
  - Marshalling the results from the servant to be returned to the client.



RMI Coding

Useful link: https://blog.csdn.net/qq_28081453/article/details/83279066

Server example

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer {
    public static void main(String[] args) {
        try {
            WorldClock worldClock = new WorldClock();
            LocateRegistry.createRegistry(8888);
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("WorldClockService", worldClock);
            // remove object
            // UnicastRemoteObject.unexportObject(worldClock, false);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client example

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMIClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry();
            IWorldClock worldClock = (IWorldClock) registry.lookup("WorldClockService");
            System.out.println(worldClock.getLocalDateTime("Asia/Shanghai"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Services example

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.time.LocalDateTime;

public interface IWorldClock extends Remote {
    LocalDateTime getLocalDateTime(String zoneId) throws RemoteException;
}

import java.io.Serializable;
import java.rmi.RemoteException;
import java.time.LocalDateTime;
import java.time.ZoneId;

public class WorldClock implements IWorldClock, Serializable {
    @Override
    public LocalDateTime getLocalDateTime(String zoneId) throws RemoteException {
        return LocalDateTime.now(ZoneId.of(zoneId)).withNano(0);
    }
}
```

Commands to run

Terminal 1:

```
Cd <compiled .class files>
rmiregistry
```

Terminal 2

```
Cd <compiled .class files>
java RMIServer
```

Terminal 3

```
Cd <compiled .class files>
java RMIClient
```

Security

Why Security is so important in DS?

- **Goal**: Restrict access to information/resources to just to those entities that are authorized to access. Protect sensitive data.
- There is a pervasive need for measures to guarantee the privacy, integrity, and availability of resources in DS.
- Security attacks take various forms: Eavesdropping, masquerading, tampering, and denial of service. The system suffers security threats all the time.
- Designers of secure distributed systems must cope with the exposed interfaces and insecure network in an environment where attackers are likely to have knowledge of the algorithms used to deploy computing resources. We cannot trust conditions outside the system.
- Cryptography provides the basis for the authentication of messages as well as their secrecy and integrity. The system needs extra encryption methods.

Threats

- Security threats fall into three broad classes:
  - Leakage – the acquisition of information by unauthorized recipients;
  - Tampering – the unauthorized alteration of information;
  - Vandalism – interference with the proper operation of a system without gain to the perpetrator.

Attacks on distributed systems depend on access to an existing communication channel. A communication channel can be misused in different ways:

- Eavesdropping – obtaining copies of messages without authority.
- Masquerading – sending or receiving messages using the identity of another principal without their authority.
- Message tampering – intercepting messages and altering their contents before passing them on (or substituting a different message in their place); e.g. the man-in-the-middle attack.
- Replaying – storing intercepted messages and sending them at a later date.
- Denial of service – flooding a channel or other resource with messages in order to deny access for others.
  - Solutions
    - IP whitelist/blacklist
    - Current limiting

Worst-case assumptions and design guidelines

- Interfaces are exposed – e.g. a socket interface is open to the public, in much the same way as the front door of a house.
- Networks are insecure – messages can be looked at, copied and falsified.
- Limit the lifetime and scope of each secret – keys and passwords can be broken, given enough time and resources.
- Algorithms and program code are available to attackers – the bigger and more widely distributed a secret is, the greater the risk of its disclosure. Open source code is scrutinized by many more programmers than closed source code and this helps to find potential security problems before they are taken advantage of.
- Attackers may have access to large resources – available computing power needs to be predicted into the life time of the system and systems need to be secure against some orders of magnitude beyond this.
- Minimize the trusted base – parts of the system that are responsible for enforcing security are trusted, the greater the number of trusted parts the greater the complexity and so the greater risk of errors and misuse.

Cryptography

Familiar names for the protagonists in security protocols:

- Alice – First participant.
- Bob – Second participant.
- Carol – Participant in three- and four-party protocols.
- Dave – Participant in four-party protocols.
- Eve – Eavesdropper.
- Mallory – Malicious attacker.
- Sara – A server.

Encryption keys

There are two main classes of encryption algorithms: shared secret key algorithms and public/private key algorithms. Some common cryptographic notation includes:

- k_A – Alice's secret key.
- k_B – Bob's secret key.
- k_{AB} – Secret key shared between Alice and Bob.
- k_A^{priv} – Alice's private key (known only to Alice).
- k_A^{pub} – Alice's public key (published by Alice for everyone to read).
- $\{M\}_k$ – Message M encrypted with key k.
- $[M]_k$ – Message M signed with key k.

Basic properties

- Given an encryption algorithm, E, a decryption algorithm, D, a key, k, and a message, M, $\{M\}_k = E(M, k)$ and $M = D(\{M\}_k, k)$.
- If $k = k_A$ is Alice's secret key then $\{M\}_k$ can only be decrypted by Alice using k.
- If $k = k_{AB}$ is a secret key shared between Alice and Bob then $\{M\}_k$ can only be decrypted by Alice or Bob using k.
- If $k = k_A^{priv}$ is Alice's private key, from a public/private key pair, then $\{M\}_k$ can be decrypted by anyone who has k_A^{pub}.
- If $k = k_A^{pub}$ is Alice's public key, from a public/private key pair, then $\{M\}_k$ can only be decrypted by Alice using k_A^{priv}.
- Private/secret keys should be securely maintained since their use is compromised if an attacker obtains a copy of them.
- Symmetric advantage over asymmetric: Public/private key encryption algorithms typically require 100 to 1000 times more processing power than secret-key algorithms.

Secrecy and integrity

A fundamental policy is one of ensuring secrecy of a message. If Alice and Bob have agreed to a shared key and encryption/decryption algorithm then for a sequence of messages $M_1, M_2, \ldots$:

1 Alice uses k_{AB} and E to encrypt message Mi and sends $\{M_i\}k_{AB}$ to Bob.
2 Bob uses k_{AB} and D to decrypt message $\{M_i\}k_{AB}$.

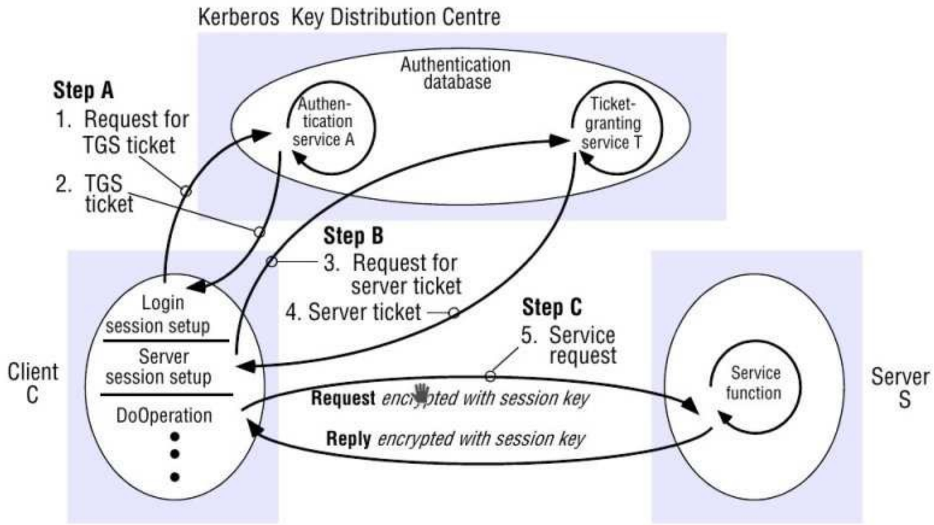
If the message makes sense when it is decrypted, or better if it contains some agreed upon value such as a checksum, then Bob can be confident that the message is from Alice and that it has not been tampered with.

Authentication using a trusted third party

Consider the case when Alice wants to access a resource held by Bob. Sara is an authentication server that is securely managed. Sara issues passwords to all users including Alice and Bob. Sara knows k_A and k_B because they are derived from the passwords.

1. Alice sends an (unencrypted) message to Sara stating her identity and requesting a ticket for access to Bob.
2. Sara sends a response to Alice encrypted using k_A consisting of a ticket encrypted in k_B and a new secret key k_{AB} for use when communicating with Bob: $\{\{Ticket\}_{kB} ,k_{AB}\}_{kA}$.
3. Alice decrypts the response using k_A. Alice cannot tamper with the ticket before sending it, because it is encrypted with k_B.
4. Alice sends a request R to Bob: $\{Ticket\}_{kB}$, Alice, R.
5. Bob receives the encrypted ticket and decrypts it using his key k_B , where the ticket is actually Ticket = $\{k_{AB}$, Alice$\}$. Alice and Bob can now communicate using the shared key or session key, k_{AB}.

System architecture of Kerberos


Kerberos Key Distribution Centre
Authentication database
Authen-tication service A
Ticket-granting service T
Step A
1. Request for TGS ticket
2. TGS ticket
Step B
3. Request for server ticket
4. Server ticket
Step C
5. Service request
Login session setup
Server session setup
DoOperation
Client C
Request encrypted with session key
Reply encrypted with session key
Service function
Server S

Challenge Response

- The use of an authentication server is practical in situations where all users are part of a single organization. It is not practical when access is required between parties that are not supervised by a single organization.
- Simple systems like Telnet send passwords from client to server “in the clear”. Such passwords are easily compromised by eavesdroppers.
- The challenge-response technique is now widely used to avoid sending passwords in the clear. The identity of a client is established by sending the client an encrypted message that only the client should be able to decrypt, this is called a challenge message. If the client cannot decrypt the challenge message then the client cannot properly respond.

Digital signature

- A digital signature serves the same role as a signature, binding an identity to a message.
- In the case of public/private keys, the identity is the public/private key pair itself.
- A digital signature requires the use of a digest. A digest is a function, Digest(M), that maps an arbitrary message, M, to a fixed length datum. The digest function must be such that a given datum is very unlikely to be mapped to from two different messages. The SHA-2 hash function is a good example of this.
- If Alice wants to sign a message, M, then Alice constructs $\{M, \text{Digest}(M)_{k_{Apriv}}\}$.
- A receiver, Bob, decrypts the digest using k_A^{pub}. Bob also computes the digest of M locally. If the message or the encrypted digest were tampered with then the results will not match.
- This is effectively a signature based on the identity k_A^{priv} since no other private key would produce that encrypted digest and no other message is likely to produce that digest. Alice cannot deny that she signed the message.

Digital certificate

A digital certificate is a document containing a statement signed by a principal. It binds information together, like a principal’s id with its public key, and is digitally signed by a certificate authority. Sara creates a digital signature by encrypting a digest of the data to be signed with her private key.

Digital signature vs digital certificate

The difference between a digital signature and a digital certificate is that the certificate binds the digital signature to the object, while the digital signature must ensure that the data or information remains secure from the moment it is sent.

Certificate chains

A series of certificates where each certificate's signature is authenticated by the subsequent certificate

- For Alice to authenticate a certificate from Sara concerning Bob’s public key, Alice must first have Sara’s public key. This poses a recursive problem.
- In the simplest case, Sara creates a self-signed certificate, which is attesting to her own public key. The self-signed certificate is widely publicized (e.g. by distributing with operating system or browser installation). This certificate is trusted. The private key must be closely guarded in order to maintain the integrity of all certificates that are signed with it.
- However, assume that Carol has signed a certificate attesting to Bob’s public key and that Sara has signed a certificate attesting to Carol’s public key. This is an example of a certificate chain. If Alice trusts Carol’s certificate then she can authenticate Bob’s identity. Otherwise Alice must first authenticate Carol’s identity using Sara’s certificate.
- Revoking a certificate is usually by using predefined expiry dates. Otherwise anyone who may make use of the certificate must be told that the certificate is to be revoked

Secure socket layer (SSL/TLS)

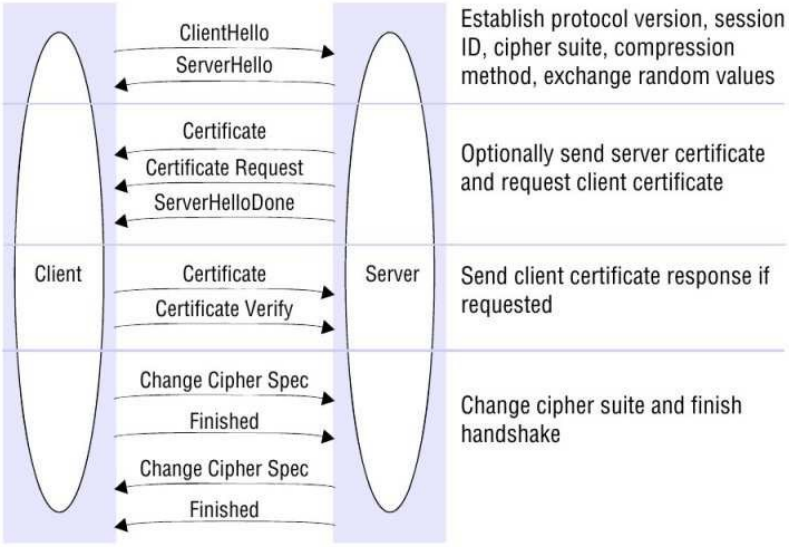
The Secure Socket Layer and its successor the Transport Layer Security (TLS) protocol are intended to provide a flexible means for clients and server to communicate using a secure channel.

There are three basic phases:

- Peer negotiation for algorithm support.
- Public key encryption-based key exchange and certificate-based authentication.
- Symmetric cipher-based traffic encryption.

Generally the protocol uses a Record Protocol layer that exchanges records; each record can be optionally compressed, encrypted and packed with a message authentication code (a signature that uses a shared secret key). Each record has a type that specifies an upper level protocol including Handshake, Change Cipher Spec and Alert Protocol.

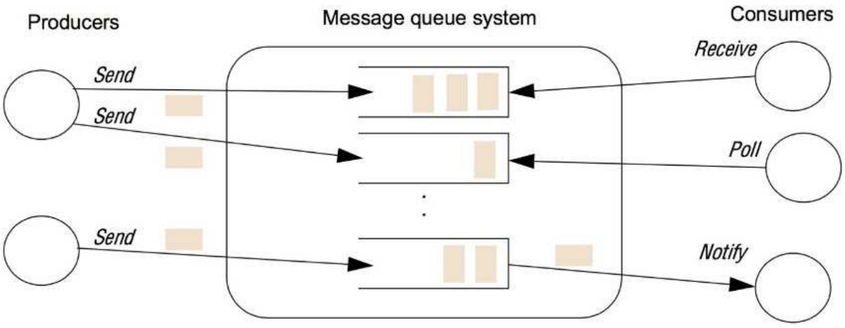
TLS handshake


Client
Server
ClientHello
ServerHello
Establish protocol version, session ID, cipher suite, compression method, exchange random values
Certificate
Certificate Request
ServerHelloDone
Optionally send server certificate and request client certificate
Certificate
Certificate Verify
Send client certificate response if requested
Change Cipher Spec
Finished
Change Cipher Spec
Finished
Change cipher suite and finish handshake

Indirect Communication

- Coupling: the dependency between objects, the higher the coupling the more cost to maintain the program. Therefore design should minimise coupling between objects and entities.
- Direct IPC are the underlying primitives – relatively low-level (perhaps the lowest level) of support for communication between processes in a distributed system, e.g. shared memory, sockets, multicast communication
- Remote invocation – based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method, e.g. request-reply protocols, remote procedure calls, remote method invocation
- Whereas direct communication is communication that takes place directly between the communicating processes, indirect communication is defined as communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s).
  - Space uncoupling – sender does not know or need to know the identity of the receiver(s)
  - Time uncoupling – sender and receiver can have independent lifetimes, they do not need to exist at the same time
  - Achieve by using middleware! Sender <—> middleware <—> receiver

Message Queues

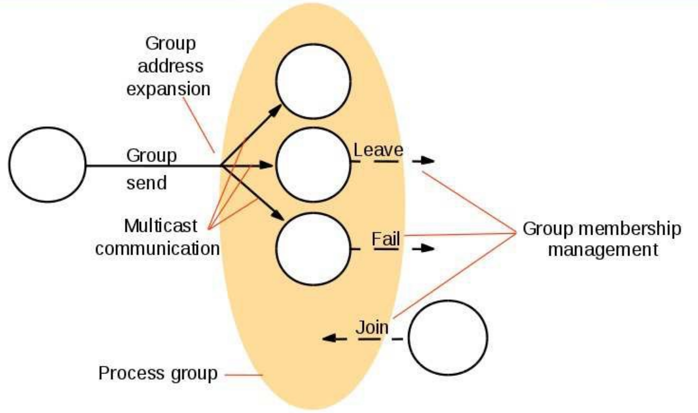


- Message queues provide a point-to-point service using the concepts of a message for data encapsulation and queue as an indirection.
- They are point-to-point in that each message is sent by a single process – producer – and received by a single process – consumer.
- Note that the producer can send to multiple message queues, i.e. point to multi-point, but consumers are bind with one message queue
- send – producers put a message on a particular queue, may block the sender if the queue has finite capacity.
- blocking receive – a consumer waits for at least one message on a queue and then returns.
- non-blocking receive – or poll, a consumer will check and get a message if there, otherwise it returns without a message.
- notify – a signal is sent to the consumer when messages are available on the queue for consumption.
- Disadvantage: more complexity, make the system more fragile

Group Communication

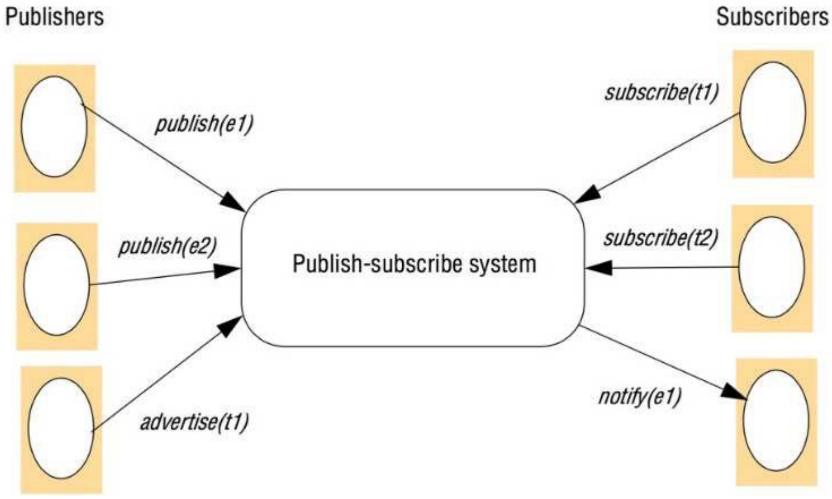
Group communication offers a space uncoupled service whereby a message is sent to a group and then this message is delivered to all members of the group.
It provides more than a primitive IP multicast:
- manages group membership
- detects failures and provides reliability and ordering guarantees

Efficient sending to multiple receivers, instead of multiple independent send operations, is an essential feature of group communication.

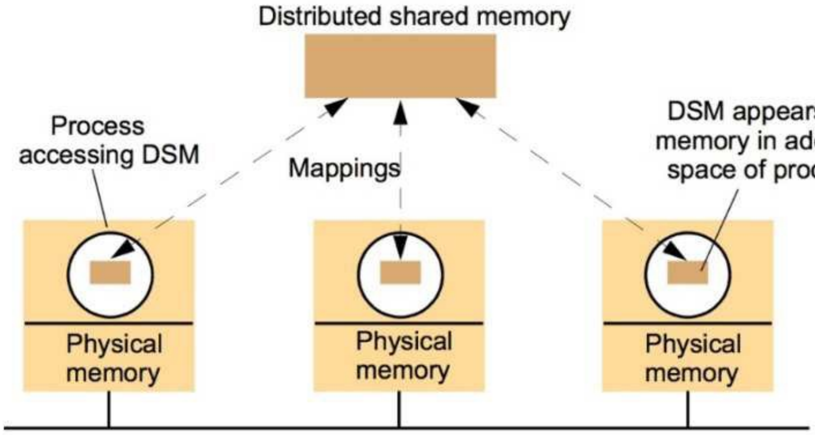


Publish/Subscribe Systems

- Publish/subscribe systems are sometimes referred to as distributed event-based systems. A publish/subscribe system is a system where publishers (event sources) publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns or query expressions over the structured events.
- Types of pub-sub systems include:
  - Channel Based – Publishers publish to named channels and subscribers subscribe to all events on a named channel.
  - Type Based – Subscribers register interest in types of events and notifications occur when particular types of events occur.
  - Topic Based – Subscribers register interest in particular topics and notifications occur when any information related to the topic arrives. (Only for developers)
  - Content Based – This is the most flexible of the schemes. Subscribers can specify interest is particular values or ranges of values for multiple attributes. Notifications are based on matching the attribute specification criteria.
  - Type vs topic: topic is queue entity, type is property of events. e.g. a YouTuber (channel) post a video, with tags #java (type), this video is sent to all non-human subscribers (topic) such as data platform, log platform, data platform also want to know about all videos.
- When and event matches a subscriber's subscription then the system sends a notification that contains the event to the subscriber.
- event: A change of state that is of interest.
- Notification: Information regarding an event that is sent to a subscriber.
- Subscriber: Express interest in particular events and get notified when they occur.
- Publisher: Publish/send structured events to an event service.
- Advertise provides an additional mechanism for publishers to declare the nature of future events, i.e. the types of events of interest that may occur. In example is declaring a type nature.

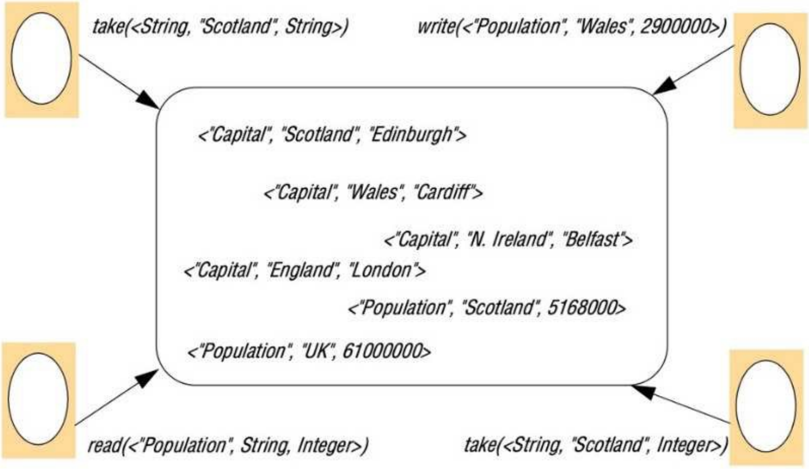


Shared memory


Distributed shared memory
Process accessing DSM
Mappings
DSM appears as memory in address space of process
Physical memory
Physical memory
Physical memory

Distributed shared memory is an abstraction for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space.
Disadvantage: accessing shared memory take much longer time than accessing local memory

Tuple Spaces


take(<String, "Scotland", String>)
write(<"Population", "Wales", 2900000>)
<"Capital", "Scotland", "Edinburgh">
<"Capital", "Wales", "Cardiff">
<"Capital", "N. Ireland", "Belfast">
<"Capital", "England", "London">
<"Population", "Scotland", 5168000>
<"Population", "UK", 61000000>
read(<"Population", String, Integer>)
take(<String, "Scotland", Integer>)

The tuple space is a more abstract form of shared memory, compared to DSM.

Distributed File Systems

Consists of three modules:

- *Flat file service*: The flat file service is concerned with implementing operations on the contents of files. A *unique file identifier* (UFID) is given to the flat file service to refer to the file to be operated on. The UFID is unique over all the files in the distributed system. The flat file service creates a new UFID for each new file that it creates. It provides interfaces to: read, write, create, delete file and get/set attributes of files.
- *Directory service*: The directory service provides a mapping between text names and their UFIDs. The directory service creates directories and can add and delete files from the directories. The directory service is itself a client of the flat file service since the directory files are stored there. It provides interfaces to lookup file, add/remove file name to/from directory, get names by pattern
- *Client module*: The client module integrates the directory service and flat file service to provide whatever application programming interface is expected by the application programs. The client module maintains a list of available file servers. It can also cache data in order to improve performance.

Distributed File Systems Requirements

Transparency :

- *Access transparency* – Client programs should be unaware of the distribution of files. Same API is used for accessing local and remote files and so programs written to operate on local files can, unchanged, operate on remote files.
- *Location transparency* – Client programs should see a uniform file name space; the names of files should be consistent regardless of where the files are actually stored and where the clients are accessing them from.
- *Mobility transparency* – Client programs and client administration services do not need to change when the files are moved from one place to another.
- *Performance transparency* – Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.
- *Scaling transparency* – The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Concurrent file updates : Multiple clients' updates to files should not interfere with each other. Policies should bemanageable.

File replication : Each file can have multiple copies distributed over several servers, that provides better capacity for accessing the file and better fault tolerance.

Hardware and operating system heterogeneity : The service should not require the client or server to have specific hardware or operating system dependencies.

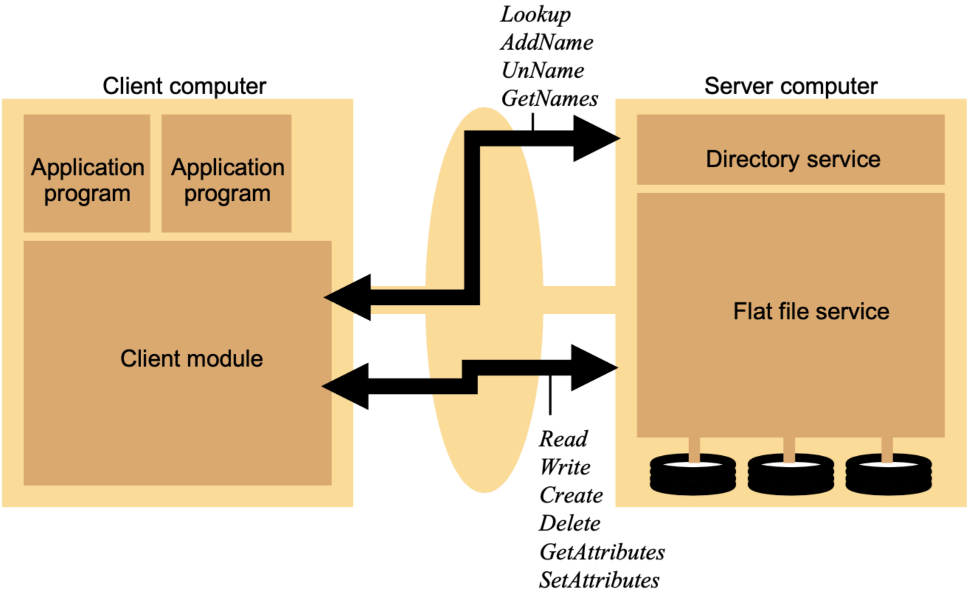
Fault tolerance: Transient communication problems should not lead to file corruption. Servers can use at-most-once invocation semantics or the simpler at-least-once semantics with *idempotent* operations. Servers can also be *stateless*.

Consistency : Multiple, possibly concurrent, access to a file should see a consistent representation of that file, i.e. differences in the files location or update latencies should not lead to the file looking different at different times. File meta data should be consistently represented on all clients.

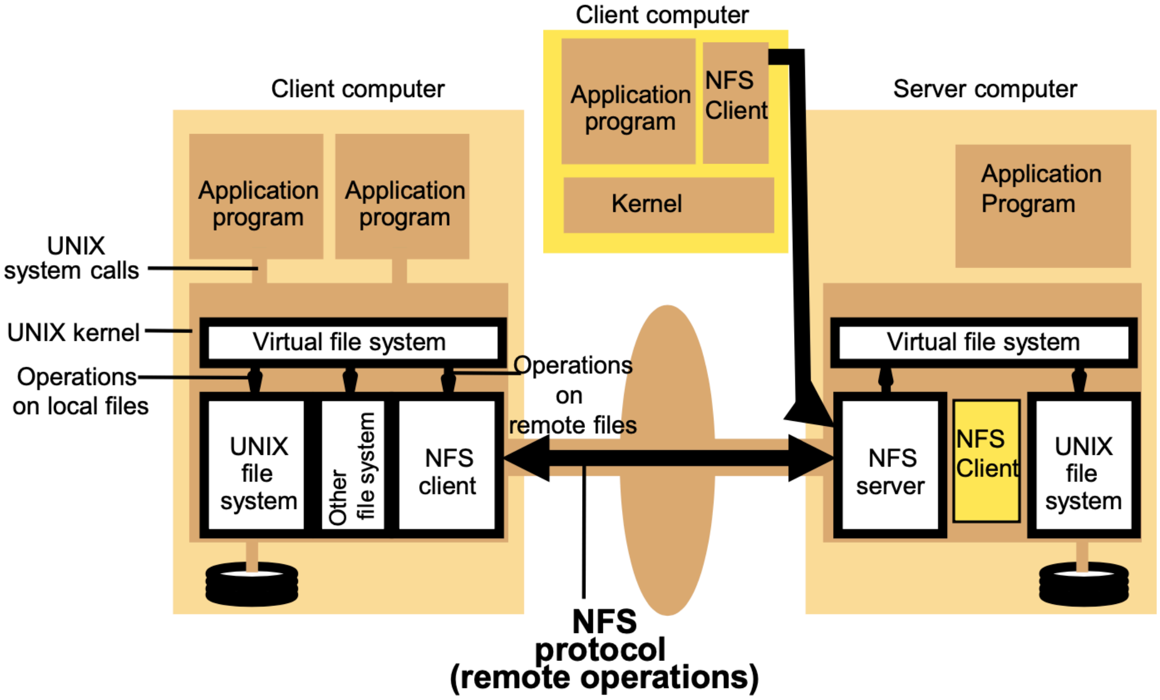
Security : Client requests should be authenticated and data transfer should be encrypted.

Efficiency : Should be of a comparable level of performance to conventional file systems.

File Service Architecture (FSA)

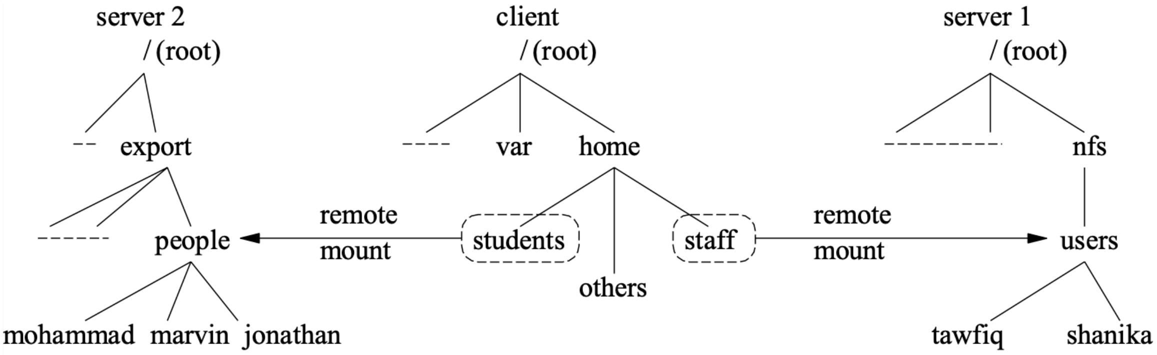


Sun Network File System (NFS)



Mount service

- File system essentially is a file
- For a specific file system, find the file system, and expand the path as a tree structure. The file system store the relationship between path in disk and file.

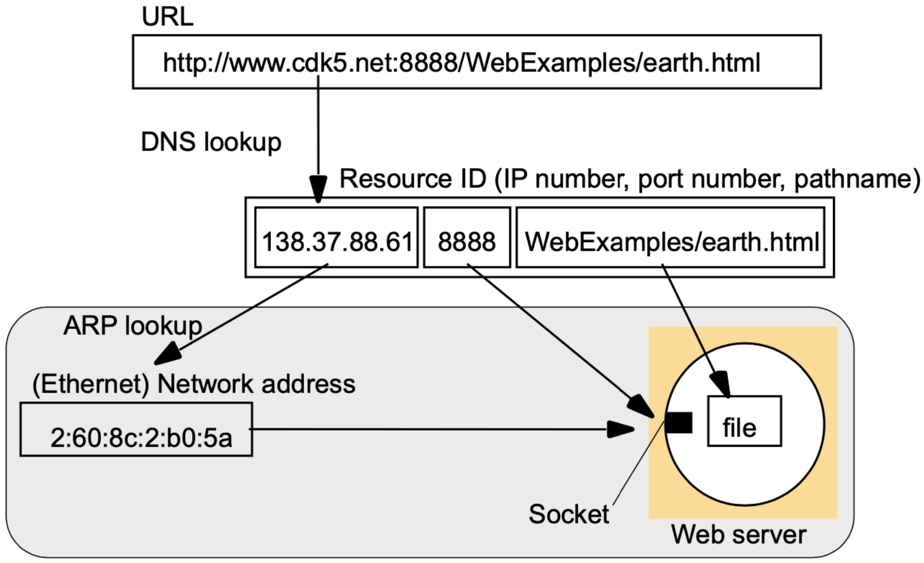


Naming services

- In a distributed system, names are used to refer to a wide variety of resources such as:
  - Computers, services, remote objects, and files, as well as users.
- **Goal:** facilitates communication and resource sharing.
  - A name in the form of URL is needed to access a specific web page.
  - Processes cannot share resources managed by a computer system unless they can name them consistently.
  - Users cannot communicate within one another via a DS unless they can name one another, with email address.

Benefits of naming services: resource localisation, uniform naming, device independent address

Accessing Resources from URL


URL
http://www.cdk5.net:8888/WebExamples/earth.html
DNS lookup
Resource ID (IP number, port number, pathname)
138.37.88.61
8888
WebExamples/earth.html
ARP lookup
(Ethernet) Network address
2:60:8c:2:b0:5a
file
Socket
Web server

URI, URL, URN

- Currently, different name systems are used for each type of resource:

resource	name	identifies
file	pathname	file within a given file system
process	process	process on a given computer
port	port number	IP port on a given computer

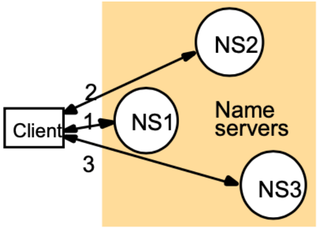
- Uniform Resource Identifiers (URI) offer a general solution for any type of resource. There two main classes:
  - Uniform Resource Locator (URL)
    typed by the protocol field (http, ftp, nfs, etc.) part of the name is service-specific resources cannot be moved between domains
  - Uniform Resource Name (URN)
    requires a universal resource name lookup service - a DNS-like system for all resources

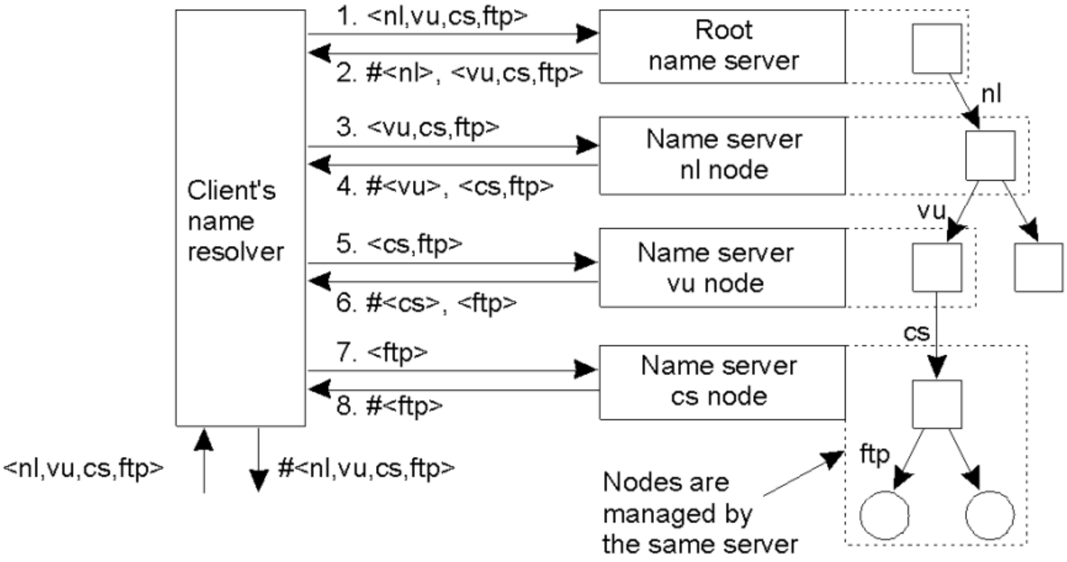
Navigation

- Navigation is the act of chaining multiple Naming Services in order to resolve a single name to the corresponding resource.
- Namespaces allows for structure in names.
- URLs provide a default structure that decompose the location of a resource in
  - protocol used for retrieval
  - Internet end point of the service exposing the resource
  - service specific path
- This decomposition facilitates the resolution of the name into the corresponding resource
- Moreover, structured namespaces allows for iterative navigation...

Iterative Navigation

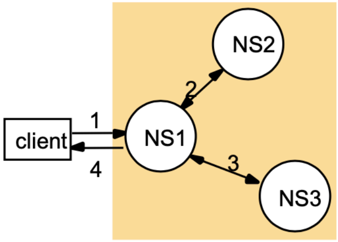
- Used in DNS & NFS
- Client contacts one NS each time
- Name Server either resolves name, or suggests other name server to contact
- Resolution continues until name resolved or name found to be unbound



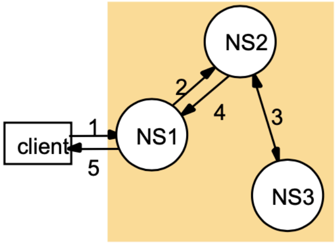


Server controlled navigation

- In an alternative model, name server coordinates naming resolution and returns the results to the client. It can be:
  - Recursive:
    - it is performed by the naming server
    - the server becomes like a client for the next server
    - this is necessary in case of client connectivity constraints
  - Non recursive:
    - it is performed by the client or the first server
    - the server bounces back the next hop to its client



Non-recursive
server-controlled

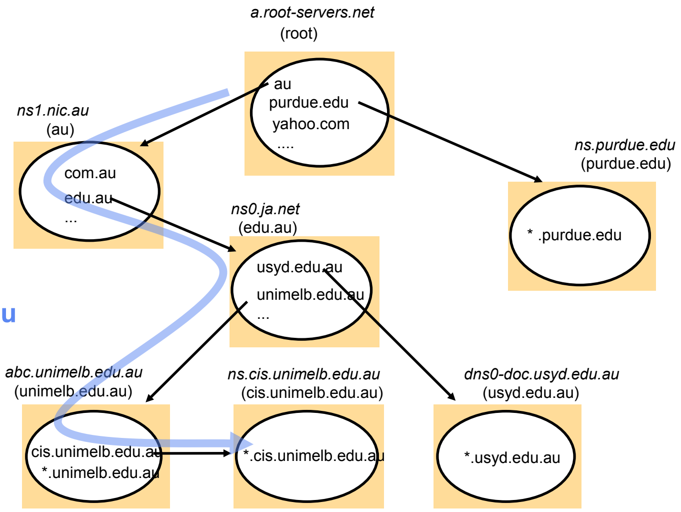


Recursive
server-controlled

- NS1 is hiding NS2 and NS3, avoid leaking their network location to the client, NS1 is acting as a proxy to hide the complex network structure.
- In non-recursive server-controlled, if NS1 don't know it will ask NS2, if NS2 don't know, NS1 will ask NS3.
- In recursive server-controlled, if NS1 don't know it will ask NS2, if NS2 don't know, NS2 will ask NS3.
- DNS offers recursive navigation as an option, but iterative is the standard technique.
- Recursive navigation must be used in domains that limit client access to their DNS information for security reasons.

DNS

- A distributed naming database (specified in RFC 1034/1035)
- Name structure reflects administrative structure of the Internet
- Rapidly resolves domain names to IP addresses
  - exploits caching heavily
  - typical query time ~100 milliseconds
- Scales to millions of computers: partitioned database, caching
- Resilient to failure of a server: replication
- Basic DNS algorithm for name resolution (domain name -> IP number):
  - Look for the name in the local cache
  - Try a superior DNS server, which responds with:
    - another recommended DNS server
    - the IP address (which may not be entirely up to date)



Note: Name server names are in italics, and the corresponding domains are in parentheses.
Arrows denote name server entries

authoritative path to lookup:
dsys.cis.unimelb.edu.au