

Distributed Systems

Lectured by Emil C Lupu

Typed by Aris Zhu Yi Qing

April 25, 2022

Contents

1	Characteristics	3	5	Remote Procedure Calls (RPCs)	7
1.1	Distribution Transparencies	3	5.1	RPC Interactions	7
1.2	Challenges	3	5.2	Stub Procedures	7
1.3	Wrong Assumptions	3	5.3	Dispatcher	8
1.4	Terminology	3	5.4	Interface Compiler	8
2	Architecture	3	5.5	Interface Type Checking	8
2.1	Layered architecture	3	5.5.1	Same Interface	8
2.2	Object-based and service-oriented architectures	4	5.5.2	Allow subtyping	8
2.3	Message-based architectures	4	5.5.3	Structural Compability	8
2.4	Peer-to-peer	4	5.6	Binding	9
3	Message-passing and IPC	4	5.7	Failure	9
4	Complex Data Representation	5	5.7.1	Best Effort (Maybe) semantics	9
4.1	Definition	5	5.7.2	At least once semantics	9
4.2	Encoding structures	5	5.7.3	At most once semantics	9
4.3	Composed structures	5	5.7.4	Zero or once (Transactional) semantics	9
4.4	Object references(pointers)	5	5.7.5	Server Failure	10
4.5	Extensible Markup Language (XML)	6	5.7.6	Client Failure	10
4.6	JSON	6	5.8	Implementation	10
4.7	Java Object Serialization	6	5.9	Concurrency	10
4.8	Liminations	7	6	Distributed Object Systems (Java RMI)	11
			6.1	Remote Interface	11
			6.2	Client Server Interaction	11
			6.3	Remote Object Implementation	11
			6.4	Server and Client Implementation	12
			6.5	Garbage Collection of Remote Objects	12

7	Time Service	12		
7.1	Cristian's Algorithm	12	9.5.3	Bastion Host 23
7.2	Berkley Algorithm	13	9.5.4	Firewall Architectures 23
7.3	Network Time Protocol (NTP)	13		
7.4	Vector Clocks	14		
8	Coordination	14		
8.1	Unreliable v.s. Reliable Failure Detectors	14		
8.2	Mutual Exclusion	14		
8.2.1	Properties and Performance	14		
8.2.2	Central Server Algorithm	15		
8.2.3	Ring-based Algorithm	15		
8.2.4	Ricart & Agrawala	15		
8.3	Elections	16		
8.3.1	Ring Elections	16		
8.3.2	Bully	16		
8.4	Consensus	17		
8.4.1	The Byzantine General Problem	17		
8.4.2	Unsigned Messages (UM)	18		
9	Security	18		
9.1	Introduction	18		
9.1.1	Goals	18		
9.1.2	Requirements	19		
9.2	Cryptography	19		
9.2.1	Definition	19		
9.2.2	Secret Key (Symmetric) Cryptography	19		
9.2.3	Public Key (Asymmetric) Cryptography	19		
9.2.4	Message Digest	19		
9.3	Symmetric Key Distribution and Authentication	20		
9.3.1	Key Distribution Centres (KDC)	20		
9.3.2	Needham-Schroeder	20		
9.3.3	Kerberos Authentication Service (V4)	20		
9.4	Certificates and Public Key Infrastructure	21		
9.4.1	Certificates	21		
9.4.2	Pretty Good Privacy (PGP)	21		
9.5	Firewalls	22		
9.5.1	Components	22		
9.5.2	Packet Filtering Rules	22		

1 Characteristics

1.1 Distribution Transparencies

Realize a coherent system by *hiding distribution* from the user where possible.

- **Access:** uniform access whether local or remote
- **Location:** access without knowledge of location
- **Concurrency:** sharing without interference (requires synchronization)
- **Replication:** hides use of redundancy (e.g. for fault tolerance)
- **Failure:** conceal failures by replication or recovery
- **Migration:** hides migration of components (e.g. for load balancing)
- **Performance:** hide performance variations (e.g. through use of scheduling and reconfiguration)
- **Scaling:** permits expansion by adding more resources (e.g. cloud)

1.2 Challenges

- **Heterogeneity:** different OS, data representation, implementations, etc.
- **Openness:** need to define *interfaces* for components to easily scale up systems
- **Security:** control access to preserve integrity and confidentiality
- **Concurrency:** inconsistencies may arise with interleaving requests
- **Failure handling:** transient/permanent failures could occur at any time. It is difficult detect them and to maintain consistency.
- **Scalability:** size of the system makes it difficult to maintain information about *system state*.

1.3 Wrong Assumptions

- The network is reliable, secure & homogeneous.
- The topology does not change.
- The latency is zero.
- The bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

1.4 Terminology

- **Client:** an entity initiating an interaction
- **Server:** a component responds to interactions usually implemented as a process
- **Service:** a component of a computer system that manages a collection of resources and presents their functionality to users.
- **Middleware:** software layer between the application and the OS masking the heterogeneity of the underlying system.

2 Architecture

2.1 Layered architecture

- e.g. Network stack. Control flows downwards, results flow upwards.
- + framework is simple and easy to learn and implement
- + reduced dependency due to layer separation
- + testing is easier with such modularity
- + cost overheads are fairly low
- scalability is difficult due to fixed framework structure
- difficult to maintain, since a change in a single layer can affect the entire system because it operates as a single unit
- parallel processing is not possible

2.2 Object-based and service-oriented architectures

- e.g. RMI
- + reusability, easy maintainability and greater reliability due to modularity
- + improved scalability and availability: multiple instances of a single service can run on different servers at the same time.
- Increased overhead: Service interactions require validations of inputs, thereby increasing the response time and machine load, and reducing the overall performance

2.3 Message-based architectures

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct process messaging	Messaging via mailbox
Referentially decoupled	Event-based (publish-subscribe)	Shared data spaces

- Referentially coupled: processes name sender/receiver in their communication.
- Temporally coupled: both sender and receiver need to be up and running.

2.4 Peer-to-peer

- structured: Each node is indexed so that the location is known, and messages are routed according to the topology.
- unstructured: *flooding* or *random walks* or both.
- + no server needed since individual workstations are used to access files
- + resilient to computer failures, since it does not disrupt any other part of the network
- + very scalable

- poor performance with larger networks since each computer is being accessed by other users
- no central file system, hard to look up or backup
- ensuring that viruses are not introduced into the network is the responsibility of each individual user.
- There is no security other than assigning permissions.

3 Message-passing and IPC

- Asynchronous send: sender continues its execution once the message has been copied out of its address space
 - + mostly used with *blocked receive*
 - + underlying system must provide buffering for receiving messages independently of receiver processes
 - + *loose* coupling: sender does not know when message will be received, does not suspend execution until the message has been received
 - *Buffer exhaustion* (no flow control)
 - formal verification is more difficult, as need to account for the state of the buffers
- Synchronous send: *blocked send*, where the sender is held up until actual receipt of the message by the destination.
 - + usually used with blocking receive, where receiver execution is suspended until a message is received.
 - + synchronization between sender and receiver
 - + generally easier to formally reason about synchronous systems
 - what if no receivers? message loss?
 - No multi-destination, requiring synchronization with all receivers.
 - implementation more complicated
 - The underlying communication service is expected to be *reliable*, i.e. to guarantee in order message delivery.

- Asynchronous receive: process continues execution if there are no messages. hardly provided as primitives
- Blocked receive: the destination process blocks if no message is available, and receives it into a target variable when available.
- Please check the coursework for how UDP client/server is implemented in Java, e.g. how datagram, socket, port are used.

4 Complex Data Representation

4.1 Definition

- **Marshalling** takes a collection of data items and transform them in a format suitable for transmission.
- **Unmarshalling** reconstitutes the data values and data structures from the bytes received.

4.2 Encoding structures

Constructed types need to be flattened for transfer.

```
struct Person {
    string Name;
    string place;
    unsigned long year;
}
```

e.g., {"Smith", "London", 1984}

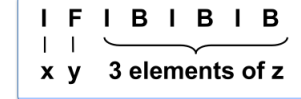
Alignment on multiples of 4 bytes.
Variable length structures are "padded" to preserve the alignment.

index	4 byte	
0-3	5	<i>length</i>
4-7	"Smit"	
7-11	"h__"	
12-15	6	<i>length</i>
16-19	"Lond"	
20-23	"on__"	
24-27	1984	<i>unsigned long</i>

4.3 Composed structures

```
struct rec {
    int a;
    boolean b;
};
struct form {
    int x;
    float y;
    rec z [ 3]; /* assume 3 elements */
};
form obj = (5, 23.75, 10, true, 5, false, 7, true)
```

can be "flattened" for transfer:
where I = int, F = float, B = boolean



4.4 Object references(pointers)

References have no meaning in the receiver's memory space. So the entire data structure pointed at must be encoded and transmitted. Structural information must be maintained and encoded in a linear message.

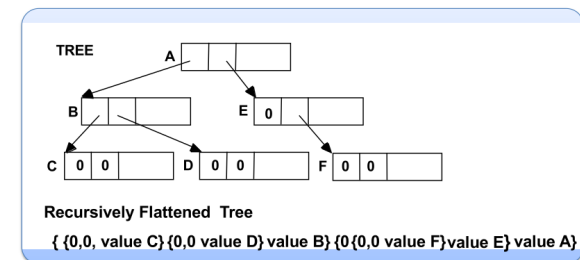


Figure 1: Reference within the object

Structural information must be flattened:

- But must not copy data multiple times.
- Number sub-objects
- Transform pointers into *handles* (ie. number) of sub-objects.

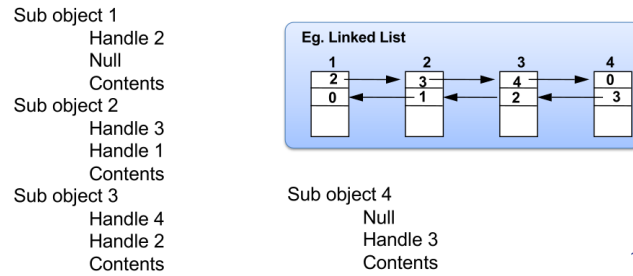


Figure 2: Reference to objects already transmitted

4.5 Extensible Markup Language (XML)

- **element**: container for data, enclosed by start and end tag. can contain other elements.
- **attribute**: used to label data — usually name/value
- **namespace**: used to scope names
 - defining a set of names each for a collection of element types and attributes referenced by a url
 - specify namespace by `xmlns` attributes
 - can use namespace name as prefix for names
- **schema**: defines elements and attributes that can appear in a document

4.6 JSON

- structural tokens: `[{] : ,`
- literal name tokens: `true, false, null`
- value: `object, array, number, string, true, false, null.`

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
  <pers:name> Smith </pers:name>
  <pers:place> London </pers:place >
  <pers:year> 1934 </pers:year>
</person>
```

Annotations in the XML code: `xmlns:pers` is labeled "Namespace attribute" and `pers:` is labeled "Namespace prefix".

Figure 3: an example for XML

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
  <xsd:element name= "person" type = "personType" />
  <xsd:complexType name="personType">
    <xsd:sequence>
      <xsd:element name = "name" type="xs:string"/>
      <xsd:element name = "place" type="xs:string"/>
      <xsd:element name = "year" type="xs:positiveInteger"/>
    </xsd:sequence>
    <xsd:attribute name= "id" type = "xs:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
```

Figure 4: an example for XML Schema

- object: `{ string : value, string : value, ... }`
- array: `[value, value, ...]`
- string: `"⟨sequence of Unicode character⟩"` with usual escapes

4.7 Java Object Serialization

- Type (class) information is included with the serialization.
- Reference to other objects are treated as *handles*.
- **Reflection**: the ability to query a class for the name and types of its attributes and methods
- Reflection allows for generic code for marshalling and unmarshalling.

```

{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": 100
    },
    "Animated" : false,
    "IDs": [116, 943, 234, 38793]
  }
}

```

Figure 5: an example for JSON

```

ByteArrayOutputStream byteobj =
    new ByteArrayOutputStream();
ObjectOutputStream out =
    new ObjectOutputStream(byteobj);
out.writeObject(new Person("Joe", "Paris", 2003));
out.close();
byteobj.close();

byte[] buffer = byteobj.toByteArray();

ByteArrayInputStream inputobj = new ByteArrayInputStream(buffer);
ObjectInputStream in = new ObjectInputStream(inputobj);
Person p = (Person) in.readObject();
in.close();
inputobj.close();

```

Figure 6: an example for Java Object Serialization

- steps of serialization

1. write class information
2. write types and names of instance variables
3. if instance variables are of a new class, then repeat the above two steps for those variables
4. uses serialization

4.8 Liminations

- Representations can have similar syntax but different meaning
 - e.g. rectangular or polar coordinates, transformation is application dependent
- Type may have no meaning outside own context
 - e.g pointer, file name
- Procedures passed as parameters
 - cannot always transfer code to different computer for execution

5 Remote Procedure Calls (RPCs)

5.1 RPC Interactions

1. Client is suspended until the call completes.
2. Request must include name of the operation and parameters passed *by value*.
3. Server operates locally and sends the result (in one or multiple messages).
4. Client decodes result and returns it to the calling procedure, which then continues processing.

5.2 Stub Procedures

- Client-side definition: the implementation of encoding parameters, send request messages, wait for reply and decode the reply and return.
- Server-side definition: the implementation of receive request, identify local procedure, decode parameters, call the procedure, encode and send the result.

- what **stubs** do in general:
 - parameter marshallng (packing)
 - unmarshal (unpack) received messages and assign values to parameters
 - transform data representations if necessary
 - access communication primitives to send/receive messages.
- Stubs can be automatically generated from an *interface specification*.

5.3 Dispatcher

It maps incoming calls onto relevant procedure (stub). Dispatcher at server receives all “call” messages and uses procedure number (name) to identify called procedure. Upon receiving requests from client, the server

1. unmarshalls method object
2. uses method info to unmarshall arguments
3. converts remote object reference to local object reference
4. calls method object’s invoke method supplying local object reference and arguments
5. when method executed, marshalls result or exceptions into reply message and sends it back to client

5.4 Interface Compiler

- generates a number for each procedure in interface – inserted into call message by client stub procedure.
- generates the stub code and the skeleton code which can then be compiled with the client and the server.
- needs to be specified by the **Interface Definition Language (IDL)** to
 - define the types that can be used
 - define the interfaces and procedures that can be called

- define the direction of the parameters: in, out, inout
- mappings to specific languages
- e.g. the following

```
interface calc {
    void mult ( [in] float a, [in] float b, [out] float Res );
    void square ( [in] float a, [out] float Res );
}
```

5.5 Interface Type Checking

5.5.1 Same Interface

- Identity can be specified, e.g.
 - checksum over source
 - name + timestamp of last modification or compilation
- Client and server hold identity of interface.
- While connecting, check type identities are equal.
- This provides **strong type compatibility**.

5.5.2 Allow subtyping

Permit server to be subtype of client interface, i.e. provides additional operations which are not used by client, but must not modify operations in original interface.

5.5.3 Structural Compatibility

Maintain run-time representation of interface and check for structural compatibility when client connects to server. The two interfaces shown in Figure 7 are structurally equivalent. This provides **weak type compatibility**.


```

interface A {
    opa1 (in string a1,
          in short a2 , out long a4);
    opa2 (in string a4);
}

interface B {
    opb1 (in string b1,
          in short b2 , out long b3);
    opb2 (in string b4)
}

```

Figure 7: weak type compatibility example

5.6 Binding

- definition:
 - connecting to a specific server
 - assignment of a reference value (e.g. address or object reference) to a placeholder (e.g. message port or object reference variable)
- **Name server** (or **directory server**) is used to register exported interfaces and is queried to locate a server when an interface is imported.
 - When a server starts it *exports* a reference to the interface to the name server.
 - When a client wants to use a service it connects to the name server and *imports* a reference to the server.
- Please refer to the coursework code to see how Java RMI binds client with server.

5.7 Failure

5.7.1 Best Effort (Maybe) semantics

As shown in the following, there is *no fault tolerance measures*.

```

bool call (request, reply) {
    send(request);
    return receive(reply,T) // return false if timeout;
}

```

The semantics is lightweight, but leaves issues of state consistency of the server, with respect to the client, up to the application programmer.

5.7.2 At least once semantics

As shown in the following, retries up to n times — if the call succeeds then procedure has been executed once or more times since duplicate messages may have been generated.

```

bool call (request, reply) {
    int retries = n;
    while(retries--){
        send(request);
        if (receive(reply,T)) return true; }
    return false; // return false if timeout
}

```

This is useful for **idempotent** server operations, i.e. multiple executions leave the same effect on server state as a single execution.

5.7.3 At most once semantics

- Guarantees that the remote procedure is either not executed or executed once.
- The server must
 - keep track of request identifiers and discard retransmitted requests that have not completed execution.
 - buffer replies and retransmit until acknowledged by the client.
 - not crash to guarantee at-most-once semantics
- It effectively achieves exactly-once semantics if no errors or exceptions have occurred.

5.7.4 Zero or once (Transactional) semantics

- Guarantees that either the procedure is completely executed or it is not executed at all.
- The server must implement an *atomic transaction for each RPC*.
 - either the state data in the server is updated permanently by an operation taking it from one consistent state to another

Retransmit request	Duplicate Filtering	Re-execute procedure or retransmit reply	Call semantics
No	N/A	N/A	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

Figure 8: Failure semantics comparison

– or it is left in its original state, if the call is aborted or a failure occurs.

- This requires ACID (Atomicity, Consistency, Isolation, Durability) properties and implemented by a *two-phase commit* type of protocol.
 - phase 1: prepares all the aspects of the transaction
 - phase 2: permanently commit or abort them

5.7.5 Server Failure

- Client needs to know server epochs to know if there is server failure leading to loss of state information in the server.
- Use `exportid` to detect failed server: when server restarts a new `exportid` is generated and exported.
- Client receives `exportid` during binding and will include it in all messages to the server.
- Dispatcher aborts calls with incorrect `exportid`.

5.7.6 Client Failure

- **Orphan executions:** result from a client crashing while the server is executing the procedure.
- Server's response will then not be acknowledged. Server either implements a form of rollback or does nothing.

- For long running procedures, to avoid wasting resources, the server may wish to be informed of client crashes so that it can abort orphan executions.

5.8 Implementation

- **TID:** a *transaction identifier* for each invocation. This includes the export identifier.
- **sn:** a message *sequence number* to detect duplicate messages and messages which follow in the sequence of invocations.
- **flag:**
 - **ack:** please acknowledge message
 - **no ack:** no acknowledgement expected
- **params:** in or out parameters as needed.
- Please see Figure 9 for an example of RPC implementation.

5.9 Concurrency

- client: no deadlocks with *callbacks* if client multi-threaded.
- server:
 - thread-per-request: dispatcher creates new thread to handle each request
 - thread pool: fixed number of threads generated at start-up, free threads are allocated to requests by the dispatcher. lower creation overhead.
 - thread-per-session: a thread is created at connection set up to process all requests from the particular client.

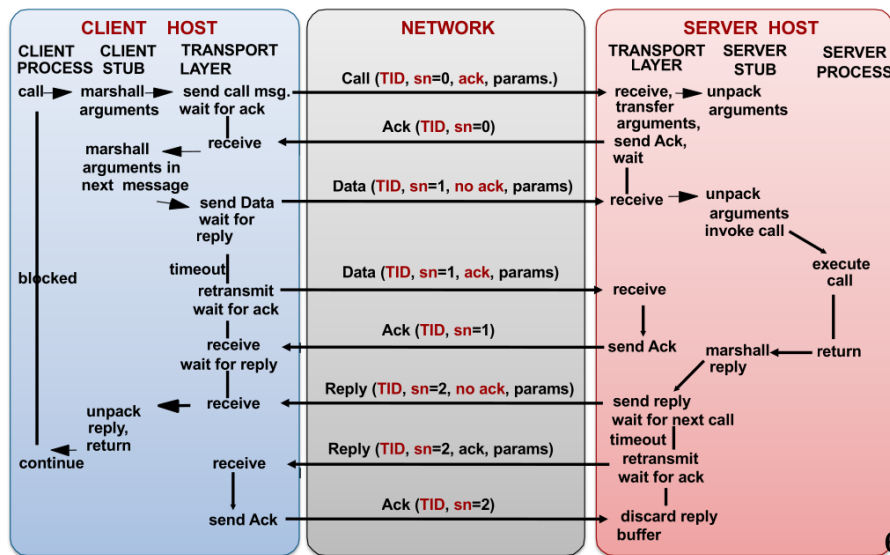


Figure 9: An example of RPC implementation

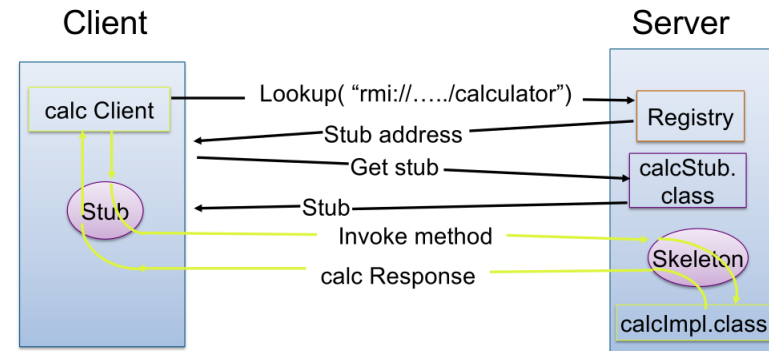


Figure 11: An example of client-server interaction

6.3 Remote Object Implementation

An object is implicitly *exported* if its class derives from the class `java.rmi.server.UnicastRemoteObject`.

6 Distributed Object Systems (Java RMI)

6.1 Remote Interface

6.2 Client Server Interaction

- relies on the ability to dynamically load code i.e. stub.
- skeleton not needed in later versions of java.

```
import java.rmi.*

public interface Calculator extends Remote {
    long add(long a, long b) throws RemoteException;
    long sub(long a, long b) throws RemoteException;
    long mul(long a, long b) throws RemoteException;
    long div(long a, long b) throws RemoteException;
}
```

Figure 10: An example of remote interface

```
import javarmi.*
public class CalculatorImpl extends UnicastRemoteObject
    implements Calculator {
    UnicastRemoteObject constructor
    exports the object.

    public CalculatorImpl() throws RemoteException {
        super();
    }

    public long add(long a, long b) throws RemoteException {
        return a + b;
    }

    public long sub(long a, long b) throws RemoteException {
        return a - b;
    }

    ...
}
```

Figure 12: An example of remote object implementation

6.4 Server and Client Implementation

- A server creates remote objects as part of mainline code.
- A server may advertise references to objects it hosts via RMI registry.
- Registry allows a binding between a URL for an object and an object reference to be queried by potential clients.
 - An object’s name is a URL formatted as “//host:port/name”.
 - Both host and port are optional.
- The server listens for incoming invocation requests which are dispatched to appropriate object.
- RMI security manager is needed in server and client if stub is loaded from server.
 - check if various operations performed by a stub are allowed
 - e.g. access to communications, files, control virtual machine, etc.

```
import java.rmi.Naming;
public class CalculatorServer {
    public static void main(String args[]) {
        if System.getSecurityManager() == null {
            System.setSecurityManager (new RMISecurityManager ());
        }
        try {
            Calculator c = new CalculatorImpl();
            Naming.rebind("rmi://localhost/CalcService", c);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
}
```

Figure 13: An example of server implementation

```
import java.rmi.*;
import Calculator;
public class CalculatorClient {

    public static void main(String[] args) {
        try {
            if System.getSecurityManager() == null {
                System.setSecurityManager (new RMISecurityManager ());
            }
            Calculator c = (Calculator) Naming.lookup(
                "rmi://remotehost/CalcService");

            System.out.println( c.sub(4, 3) );

            ... other calls ;
        } catch (RemoteException e) {
            System.out.println(" Exception:" + e);
        }
    }
}
```

Figure 14: An example of calculator client implementation

6.5 Garbage Collection of Remote Objects

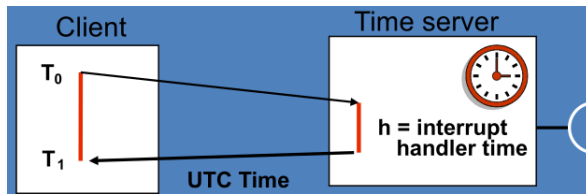
RMI runtime system automatically deletes objects no longer referenced by a client.

- When live reference enters Java VM, its reference count is incremented
- First reference sends “referenced” message to server
- After last reference discarded in client, “unreferenced” message sent to server.
- Remote object removed when no more local or remote references exist.

7 Time Service

7.1 Cristian’s Algorithm

- Assume reference time server with UTC time.

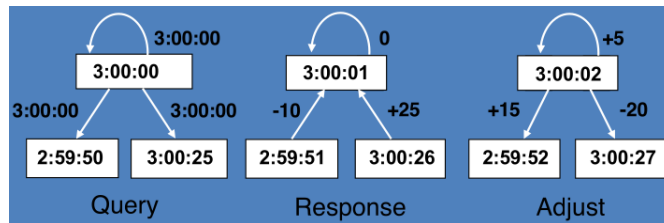


- estimated message propagation time

$$p = \frac{T_1 - T_0 - h}{2}$$

- set clock to $\text{UTC} + p$, where UTC is the time received from the server
- Measure $T_1 - T_0$ over several transactions, *remove outliers* and/or *take minimum values* as being most accurate as smaller network delays.
- Single server would be bottleneck, a single point of failure or a single point of compromise.

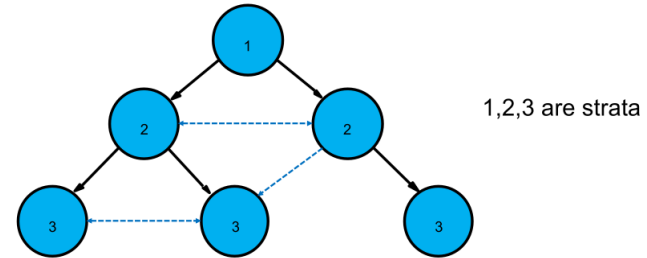
7.2 Berkley Algorithm



1. Query: **Coordinator**, chosen as master, periodically polls slaves to query clocks.
2. Response: estimates local times with compensation for propagation delay and calculates average time but ignores outliers e.g. large propagation delay
3. Adjust: sends message to each slave indicating clock adjustment.

7.3 Network Time Protocol (NTP)

- Synchronize large number of computers
- Organize servers in **strata**.



- Multiple servers across the Internet connected to UTC receivers (**primaries**).
- *Secondary* servers synchronize with primaries.
- *Tertiary* servers synchronize with secondary servers etc.
- synchronization modes
 - broadcast/multicast: one or more servers periodically multicast to other computers on *high speed LAN*. Assume small delay.
 - procedure-call/client-server: a client requests time from a few other servers, similar to Cristian's algorithm.
 - symmetric: used by master servers on LANs and layers closest to primaries, higher accuracy based on pairwise synchronization.

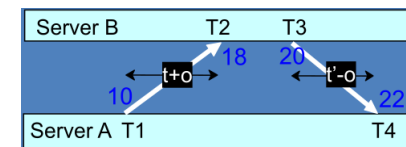


Figure 15: an example of NTP symmetric protocol

- * Using the example in Figure 15, with t and t' being the transmission delay, and o being the clock offset of B relative to A , let $a = T_2 - T_1 = t + o$, $b = T_4 - T_3 = t' - o$, we have

$$\text{RTT} = t + t' = a + b = (T_2 - T_1) + (T_4 - T_3)$$

and

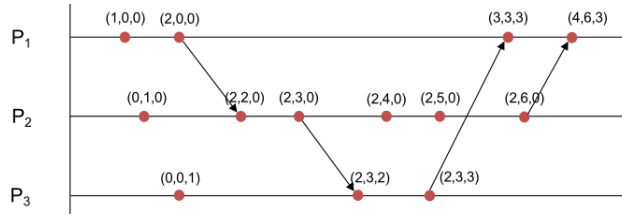
$$2o = a - b = (T_2 - T_1) - (T_4 - T_3)$$

assuming $t = t'$.

7.4 Vector Clocks

- Logical clocks are sufficient for causal ordering.
- $V_i[j]$ indicates the logical clock of process j known by process i .
- just before timestamping an event, $++ V_i[i]$.
- when it receives a timestamp t in a message, set

$$V_i[j] = \max(V_i[j], t[j]), \quad \forall j \in \{1, 2, \dots, N\}.$$



- we then have
 - if a is the event of a message m being sent from process A and b is the event of m being received by process B then $a \rightarrow b$.
 - $a \rightarrow b \iff V(a) \leq V(b)$.
 - if neither $V(a) \leq V(b)$ nor $V(a) \geq V(b)$, a and b are concurrent.
- We can define a **consistent cut** across the history of events, where the causes are present for all the effects.

8 Coordination

- **Synchronous system** means that maximum bounds can be put on: transmission delay, clock drift rate, time taken to execute an action. This means that we can detect when something has not happened.
- **Asynchronous system** means that no timing assumptions can be made. A message can take indefinite time to arrive and thus cannot distinguish between a failure and a long-term process.

8.1 Unreliable v.s. Reliable Failure Detectors

- Unreliable failure detector is not necessarily accurate, and tells whether a process is *suspected*/*unsuspected* to have failed.
- Reliable failure detector *determines* whether a process has (not) failed. If failed, not process recovery is considered.
- e.g. say process P_1 sends “ P_1 alive” message to all other process every T seconds
 - Failure detector at each process estimates *maximum* message transmission time D .
 - choice of T should reflect network delay condition
 - Expects a message every $T + D$ seconds — if received, ok; if not, suspects P_1 failed.
 - For asynchronous system, this is unreliable; for synchronous system, this is reliable.

8.2 Mutual Exclusion

8.2.1 Properties and Performance

Properties

- **Safety:** at most one process enters the critical section
- **Liveness:** requests eventually succeed, no process is blocked forever
- **Fairness:** requests are satisfied in some order in which they are sent

Performance

- minimize the bandwidth, e.g. number of messages required
- minimize synchronization delay, i.e. between one access and the following one.

8.2.2 Central Server Algorithm

1. Processes request access from central server
2. Server replies with access token, when access is granted. Other requests are queued.
3. Processes release token on finishing access.

Properties

- Safety is satisfied.
- Liveness is satisfied if each request is bounded.
- Fairness is not satisfied.

Performance

- 3 messages required for each request.
- 1 RTT required for synchronization delay. Release + Token message need to be sent.

Comments

- simple
- central point of failure/central bottleneck
- must have a designated (or elected) server

8.2.3 Ring-based Algorithm

- remove central server
- processes pass token to each other in a ring, i.e. when finished p_i sends token to $p_{i+1(\text{mod } n)}$.

- When a token is received by a process
 - it uses the token if it requires it
 - it passes the token if it doesn't

Properties

- Safety is satisfied.
- Liveness is satisfied if each request is bounded.
- Fairness: not in order of requests.

Performance

- Delay to access between 0 and N messages.
- Synchronization delay between 1 and N messages.
- Uses messages even if no one wants to use the resource.

Comments

- Variant is to embed the request time in the token and pass on tokens with earlier request times.
- Any failure breaks the ring. Needs procedures to cope with token loss and process failure in the general case.

8.2.4 Ricart & Agrawala

Please see Figure 16 for the detail.

Properties

- Safety: mutual requests are *totally ordered* so only one will receive all replies.
- Liveness: eventually all processes will release
- Fairness: order of Lamport logical timestamps

Performance

Initially

state := Released;

On Exit critical section

state := Released
reply to all queued requests

To Enter Critical Section at p_i

state := Wanted;
T := T_i // current Lamport timestamp
multicast request (T, p_i) to all processes;
wait until (N-1) replies received;
state := Held;

On receiving request (T_j, p_j) at P_i ($i \neq j$)

if (state = Held or (state = Wanted and (T, p_i) < (T_j, p_j))
 queue request (T_j, p_j);
else reply.

Figure 16: Ricart & Agrawala's algorithm

- gaining entry requires: $N - 1$ replies and 1 (if multicast support) or $N - 1$ (if no multicast support messages).
- minimum synchronization delay: 1 message

Comments

Highly dependent on reliability of processes and messages.

8.3 Elections

1. Set of processes p_i , each has a variable **elected_i**.
2. Processes can be **participant** or **non-participant**.
3. At the beginning of the election all participants set **elected_i** = null.
4. At the end of election, all participant processes must have the same value e.g. P of **elected_i**.

8.3.1 Ring Elections**Steps**

1. Initially all P_i are nonparticipants.

- P_i calls for election
- $P_i^{\text{status}} := \text{part}(\text{participant})$.
- send **election**(P_i) to $P_{i+1 \bmod n}$

2. When P_j receives **election**(ident):

- (First Round) if **ident** > P_j : forward **election**(ident), set $P_j^{\text{status}} := \text{part}$.
- (Second Round) if **ident** < $P_j \wedge P_j^{\text{status}} = \text{nonpart}$: send **election**(P_j), set $P_j^{\text{status}} := \text{part}$.
- (Third Round) if **ident** = P_j : send **elected**(P_j), $P_j^{\text{status}} = \text{nonpart}$.

3. When P_j receives **elected**(ident):

- **elected_j** = **ident**, $P_j^{\text{status}} := \text{nonpart}$.
- forward **elected**(ident), *unless* **ident** = P_j .

Analysis

- Assumptions: No failures, asynchronous, any process can begin an election.
- Safety & liveness: eventually all processes will receive and agree on the maximum if there are no failures.
- Performance: worst case $3N - 1$ messages required for single election.

8.3.2 Bully**Steps**

1. When P_i notices the coordinator is no longer there it initiates an election by sending 'ELECTION' to all P_j with $j > i$.
2. There are two cases:
 - If no 'ANSWER' received, it declares itself as the 'COORDINATOR'.

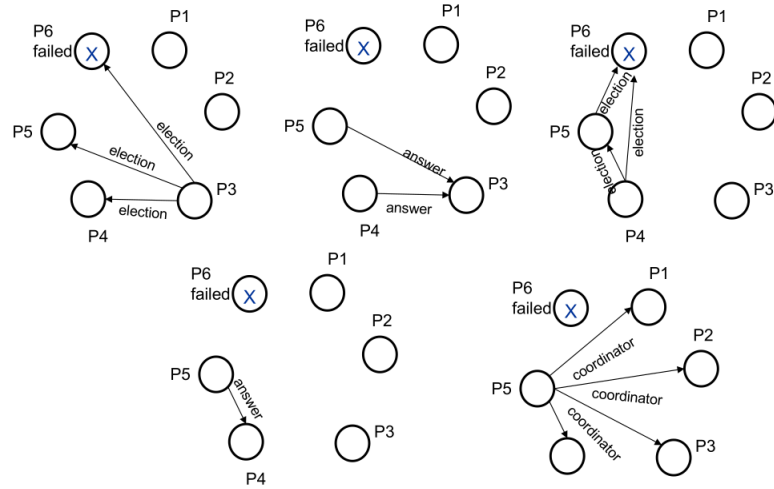


Figure 17: An example of Bully election

- If 'ANSWER' received, it waits for a 'COORDINATOR' message. If non arrives, it starts a new election. If arrives, it sets $elect_id_i$ to the value received.
3. When P_j receives 'ELECTION' it sends 'ANSWER' and starts another election, unless it has already done it.

Analysis

- Safety: No two processes can decide that they are the coordinator, assuming that failed processes cannot be restarted with the same identifier. Otherwise two processes can start again and each decide that they are the coordinator.
- Liveness: If no messages are lost all available processes will reply.
- Performance: $O(N^2)$ in the worst case.

8.4 Consensus

8.4.1 The Byzantine General Problem

A commanding general must send an order to his $n - 1$ lieutenant generals s.t.

IC1 All loyal lieutenants obey the same order.

IC2 If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

where "IC" stands for Integrity Condition. In the world of networks, command general can be mapped to the process sending the request, lieutenants can be mapped to the receiving processes.

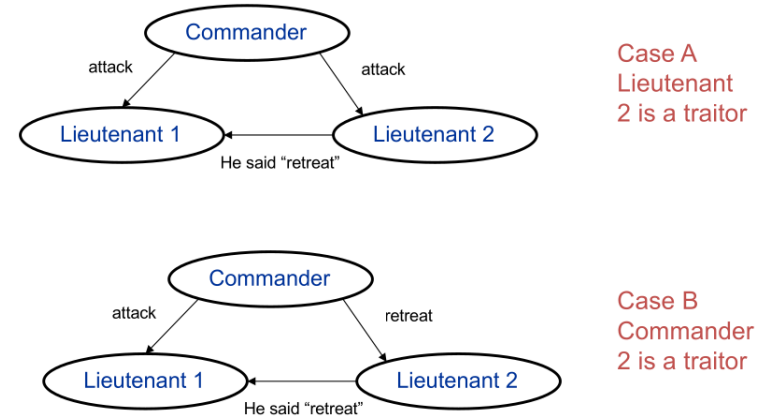


Figure 18: impossibility results

From Figure 18, we can see that in case B, if lieutenant 1 attacks he violates IC1. Lieutenant 1 cannot distinguish, from the information available to him, between case A & case B.

No Solution exists for three generals that works in the presence of a single traitor.

In general, No solution with fewer than $3m + 1$ generals can cope with m traitors.

8.4.2 Unsigned Messages (UM)

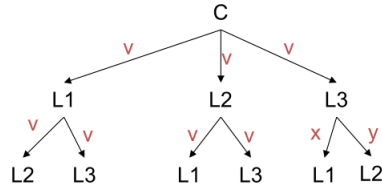
We define $UM(n, m)$ as the unsigned messages algorithm for n generals and m traitors, and assume that

- every message that is sent is delivered correctly
- the receiver of a message knows who sent it
- the absence of a message can be detected

$UM(n, 0)$, i.e. no traitors

- Commander sends v to every lieutenant
- Each lieutenant uses the value received or v_{default} if no value received.

Example $n=4, m=1, UM(4,1)$ L3 is a traitor



At the end of stage 1

L1: $v_1 = v$
L2: $v_2 = v$
L3: $v_3 = v$

At the end of stage 2

L1: $v_1 = v, v_2 = v, v_3 = x$
L2: $v_1 = v, v_2 = v, v_3 = y$
L3: $v_1 = v, v_2 = v, v_3 = v$

Each of the lieutenants arrives at the same decision and the value sent by C is the majority value

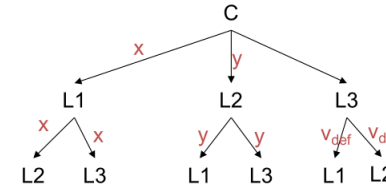
$UM(n, m)$

- For each lieutenant,
 - let v_i be the value received from commander or v_{default} if no value received

– send v_i to $n - 2$ other lieutenants using $UM(n - 1, m - 2)$

- For each i and each $j \neq i$
 - let v_i be the value lieutenant received from other lieutenants or v_{default} if no value received.
 - Lieutenant $_i$ uses $\text{majority}(v_1, v_2, \dots, v_{n-1})$.

Example $n=4, m=1, UM(4,1)$ C is a traitor



At the end of stage 1

L1: $v_1 = x$
L2: $v_2 = y$
L3: $v_3 = v_{\text{def}}$

At the end of stage 2

L1: $v_1 = x, v_2 = y, v_3 = v_{\text{def}}$
L2: $v_1 = x, v_2 = y, v_3 = v_{\text{def}}$
L3: $v_1 = x, v_2 = y, v_3 = v_{\text{def}}$

Each of the lieutenants receives the same value $\text{majority}(x, y, v_{\text{def}})$

9 Security

9.1 Introduction

9.1.1 Goals

- **Confidentiality:** prevent disclosure of info to unauthorized users + prevent analysis of traffic characteristics
- **Integrity:** prevent modification of info by unauthorized users — include no duplication, replays, insertions or reordering
- **Availability:** prevent denial of service e.g. by disruption

9.1.2 Requirements

- **Identification:** establishing the identity of the subject
- **Authentication:** validity of the identity of sender or server, e.g. passwords, biometrics, 2-factor authentication, etc.
- **Access control:** control over who has access to services or resources within the system.

9.2 Cryptography

9.2.1 Definition

- **Encryption:** a transformation of information based on substitution (table lookup) or transposition (exchanges bytes), i.e. a function E with key k ,

$$E_k[M]$$

where M is the message.

- **Decryption:** inverse of encryption to obtain original information, i.e. a function D with key k' ,

$$D_{k'}[C]$$

where C is the encrypted message.

9.2.2 Secret Key (Symmetric) Cryptography

- basis for *Data Encryption Standard* (DES)
- $E = D$ and $k = k'$

9.2.3 Public Key (Asymmetric) Cryptography

- **Public key** (K) sent out over network and stored with name servers — used by sender for encryption
- **Private key** (K^{-1}) used by recipient for decryption
- cannot deduce K^{-1} from K

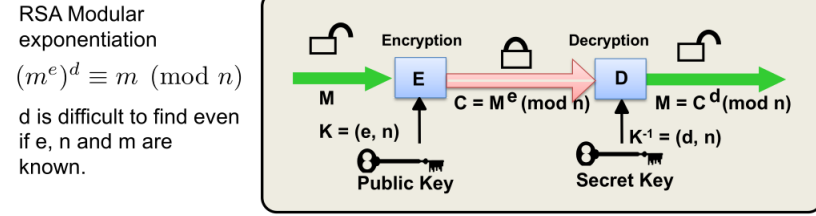


Figure 19: an illustration of authentication process

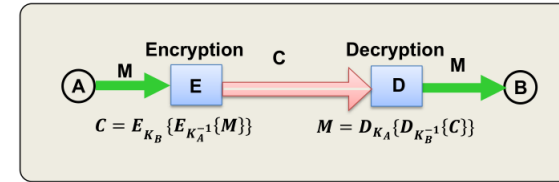


Figure 20: an illustration of signature process between B and A only

- **Authentication:** $M = D_{K^{-1}}[E_K[M]]$, a message encrypted with the public key (of the recipient) can only be decrypted with the recipient's private key.
- **Signature:** $M = D_K[E_{K^{-1}}[M]]$, a message encrypted with the private key can be decrypted with the public key of the one who encrypted it, i.e. anyone.

9.2.4 Message Digest

- Use a one-way function e.g. hash function $H(M)$, to check message M 's integrity.
- sign a message for integrity by signing a hash of it e.g. $E_{K_A^{-1}}[H(M)]$.
- send a signed message with confidentiality,

$$A \rightarrow B : \left\{ M, \{H(M)\}_{K_A^{-1}} \right\}_{K_B}.$$

9.3 Symmetric Key Distribution and Authentication

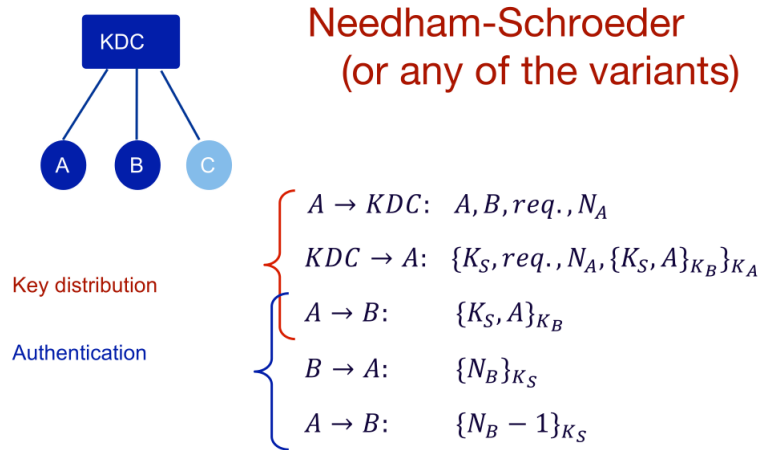
9.3.1 Key Distribution Centres (KDC)

Via a KDC, A can establish a connection with B (key distribution) by

1. $A \rightarrow \text{KDC} : \{\text{request}(K_S), B\}_{K_A}$
2. $\text{KDC} \rightarrow A : \{K_S\}_{K_A}, \{K_S\}_{K_B}$
3. $A \rightarrow B : \{K_S\}_{K_B}$

where K_S is the session key. Second step is done so to allow KDC to send one message only instead of two, thereby having less workload.

9.3.2 Needham-Schroeder



9.3.3 Kerberos Authentication Service (V4)

- Separate authentication and ticket granting service.
- Users and service providers must register with Kerberos.
- KAS stores identities, server private keys (K_S), and encrypted user passwords used to generate private user keys (K_A).

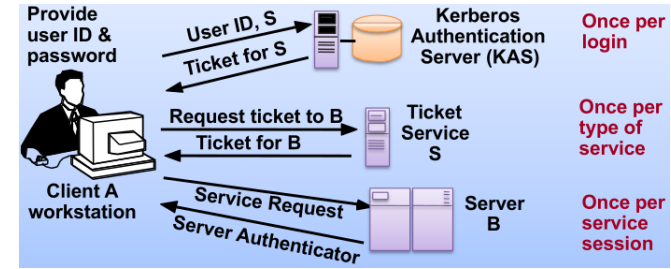


Figure 21: an overview of KAS

- Uses ticket timestamp (TS) and ticket lifetime (TL) to detect replay

Interactions

1. $A \rightarrow \text{CAS} : \text{Options, Aname, Arealm, Sname, TS}$
 - KAS generates session key K_{AS} & ticket T_{AS} for ticket server S .
 - KAS uses A 's password to generate key K_A .
2. $\text{CAS} \rightarrow A : K_A \{K_{AS}, \text{TS}_2, \text{TL}, T_{AS}\}$

- Only A can decrypt this to obtain **ticket**

$$T_{AS} = K_S \{ \text{Aname, Arealm, Aaddr, } K_{AS}, \text{TS, TL} \}$$

aimed for S to have a secure transfer of authenticated identity of A with optional authorisation data for access control.

- Uses K_{AS} to create **authenticator**

$$X_{AS} = K_{AS} \{ \text{Aname, Arealm, Appl. data, TS} \}$$

for S to prove identity of user presenting ticket is the same as that to whom ticket was issued.

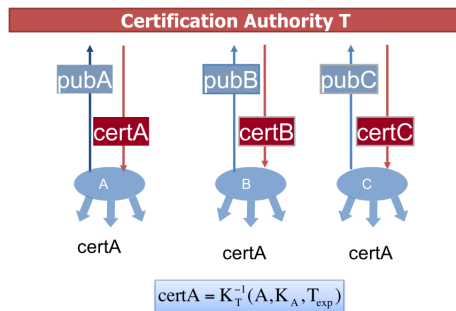
3. $A \rightarrow S : B_{\text{name}}, T_{AS}, X_{AS}$
 - Decrypt T_{AS} to get session key K_{AS} to check authenticator X_{AS} , i.e. know who is requesting access to B and verify that it is indeed A who is requesting.

- Generate session key K_{AB} and ticket T_{AB} for accessing B .
4. $S \rightarrow A: K_{AS} \{K_{AB}, B_{name}, TS, T_{AB}\}$
- Decrypt to get *ticket*
- $$T_{AB} = K_B \{Aname, Arealm, Aaddr, K_{AB}, TS, TL\}$$
- and session key K_{AB} .
- Create *authenticator*
- $$X_{AB} = K_{AB} \{Aname, Arealm, Appl. data, TS\}$$
5. $A \rightarrow B: T_{AB}, X_{AB}$
- decrypt ticket to get session key, check authenticator, generate response by adding 1 to TS in authenticator.
6. $B \rightarrow A: K_{AB} \{TS + 1\}$
- Only server B can generate this response.

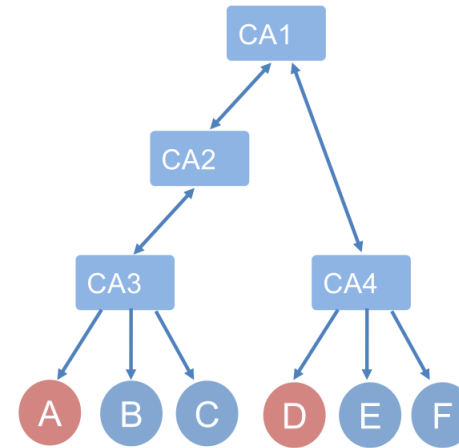
9.4 Certificates and Public Key Infrastructure

9.4.1 Certificates

- A **certificate** is a signed statement of the association between values.
- An *identity certificate* binds an identity to a cryptographic key, typically the public key of the identity.



X.509 - Certification Authority Hierarchy



Each pair (parent-child) of CAs create a certificate for each other, e.g. CA1 for CA2 and vice-versa.

Certificates can be used to build a certification path.

A needs to verify $sign_{CA4}(pubD)$
 Gets from CA Hierarchy:
 $sign_{CA3}(pubCA2)$
 $sign_{CA2}(pubCA1)$
 $sign_{CA1}(pubCA4)$
 The certificates allow A to:
 verify $pubCA2$, verify $pubCA1$,
 verify $pubCA4$, verify $pubD$

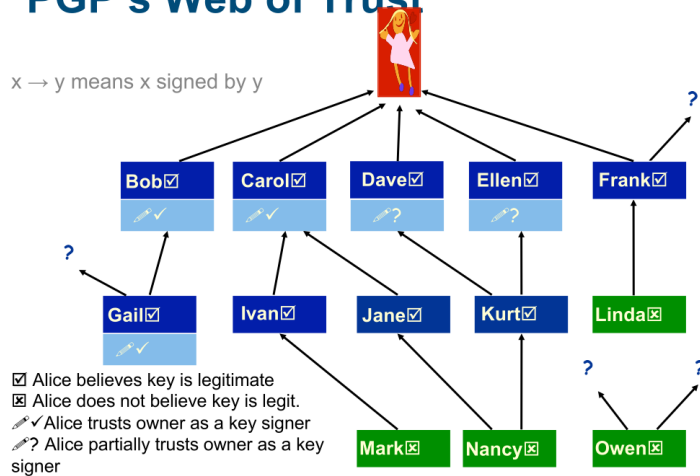
Figure 22: an example of certificate hierarchy

9.4.2 Pretty Good Privacy (PGP)

- Each user has its own **private key ring** for his own private/public key pairs.
- Public keys of other users are stored in a **public key ring**. Each entries include
 - public key
 - ID
 - $\underbrace{\text{signature}}_{\text{who else has signed the key}} + \underbrace{\text{Signature Trust Values (STV)}}_{\text{level of trust we have in the signer}}$
 - key legitimacy value: weighted sum of STVs. higher value indicates greater belief.
 - overall owner trust: level of trust we have in this key to sign others. Set by us.

- If key appears on private key ring: owner trust.
- For each signature of the public key in public key ring, PGP searches public key ring to find owner of the signature,
 - owner exists \rightarrow copy *owner trust* to *sig trust*.
 - owner doesn't exist \rightarrow unknown user.
 - STV values: undefined trust, unknown trust, partially trusted to sign, always trusted to sign, key is in “private key ring” (ultimate trust)
- Key legitimacy is calculated from STVs
 - If \exists sig-trust s.t. it is ultimate trust \rightarrow complete trust.
 - otherwise $= 1/X * \text{sum}(\text{always trusted}) + 1/Y * \text{sum}(\text{partially trusted})$
 - legitimacy values: unknown or undefined trust, key ownership not trusted, marginal trust in key ownership, complete trust in key ownership
 - owner trust may change. periodical recalculation.

PGP's Web of Trust



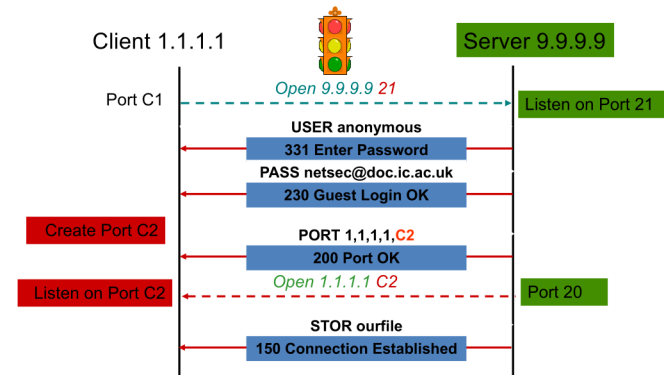
9.5 Firewalls

It is a security gateway between internal and external networks.

9.5.1 Components

- **Packet Filtering Routers:** a.k.a. *chokes*, *screening filters*. Restrict free flow of packets between networks.
- **Gateways:** applications (proxies) that provide higher-level processing e.g. authenticating users, “cleaning data”, redirecting data, etc. They usually run on **Bastion hosts**.
- end-to-end encryption between Firewalls is used to create **VPNs**.

9.5.2 Packet Filtering Rules

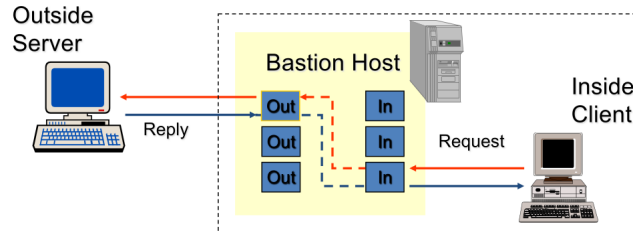


No.	Dir	Pro t.	Src Addr.	Src. Port.	Dest. Addr.	Dest. port	Flag	Action	Description
1.	Out	tcp	InsideNet	>1024	9.9.9.9	21		Allow	Allow client apps to connect to this FTP site.
2.	In	tcp	9.9.9.9	21	Insidenet	>1024	ACK	Allow	Only replies allowed
3.	In	tcp	9.9.9.9	20	Insidenet	>1024		Allow	Data connections in
4.	Out	Tcp	Insidenet	>1024	9.9.9.9	20	ACK	Allow	Return traffic

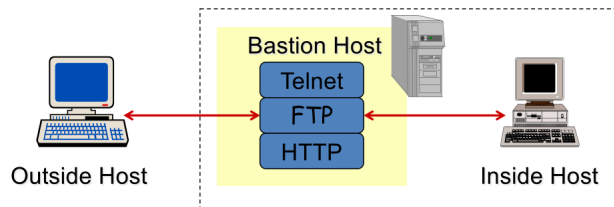
Figure 23: FTP (File Transfer Protocol) rules for FTP filtering

9.5.3 Bastion Host

- can runs *circuit-level* gateways
 - relays connections and maintains connection state
 - can drop connections based on destinations, incorrect connection packets, time, volume, etc.



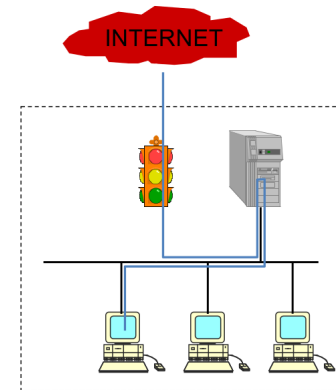
- can runs *application-level* gateways
 - application-specific (app-knowledgeable)
 - can block/filter/report based on app-level msg content
 - can scan for data leaks, viruses, etc.
 - can be configured to limit application features.
 - more processing overhead than circuit-level gateway's



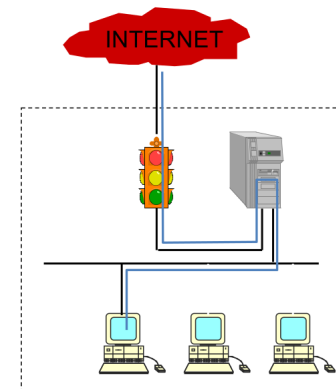
- performs auditing/accounting and administer via dedicated terminal
- should run a trusted, secure, and minimal OS
 - remove inessential applications, utility services, e.g. sed, ld, etc.
 - set file permissions, turn on file quotas, process limits etc.
 - no regular user accounts
 - no NFS mounts and make filesystems read-only if possible.

9.5.4 Firewall Architectures

- screened host architecture
 - packet filter only pass packets for which the source and destination IP address is the Bastion Host
 - if packet filter fails, packets could enter the internal network



- screened host (dual-homed) architecture
 - a host with 2 network interfaces
 - even if packet filter is skipped, still need to get pass bastion host



- screened subnet architecture
 - pass packets for which the source of destination IP address is the bastion host
 - pass packets for which the ports are for defined proxies on the gateway
 - block everything else

