Distributed System Organization

Agenda

- Software architecture
  - Layered organization
  - Object-oriented
  - SOA
  - REST
  - Publish-subscribe
  - Tuple spaces
- System architecture
  - Client-server
  - Peer-to-peer

References: Chapter 2 of "Distributed Systems by M. van Steen and A. S. Tanenbaum"

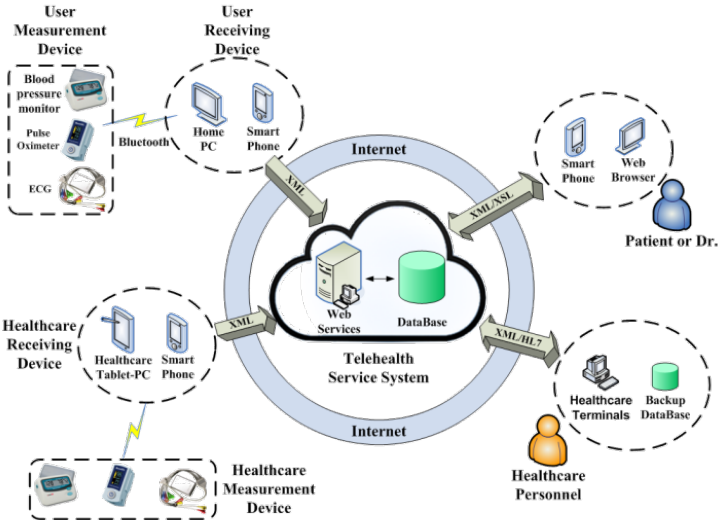
Architectures

There are two levels on how to view the organization of a distributed system

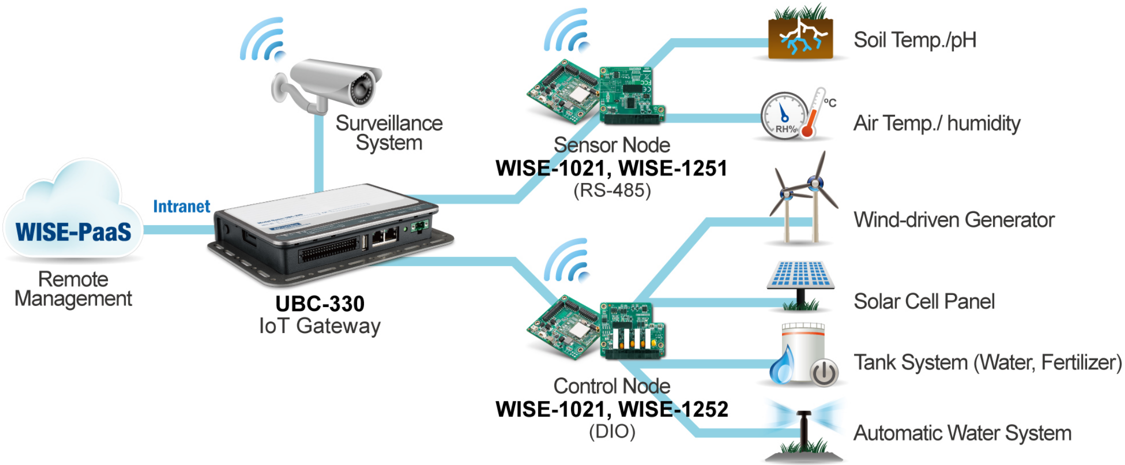
1. Software Architecture (Logical organization)
   - Identifies the components of the system
   - Describes how they are organized and how interact with each other

2. System Architecture (Physical organization)
   - Identifies the machines/devices that constitute the system and their interconnectivity
   - Describes how the software components are instantiated on the real machines

Exercise: identify the logical and physical components

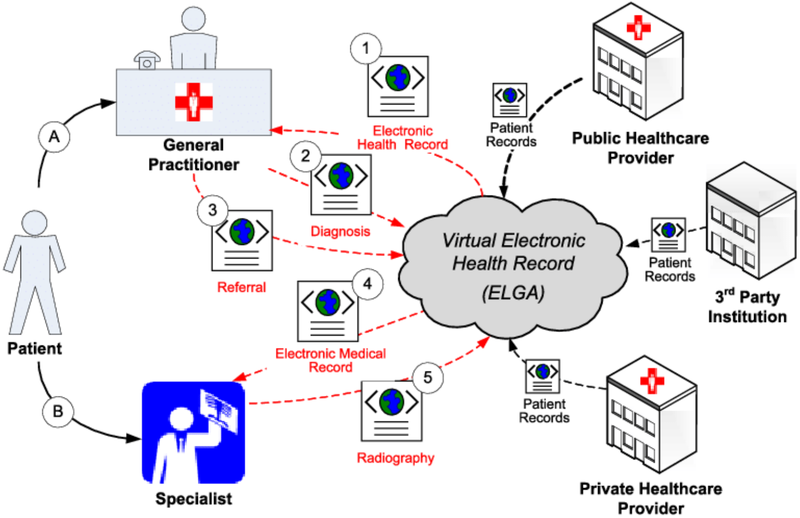


Exercise: identify the logical and physical components



Exercise: identify the logical and physical components

Exercise: identify the logical and physical components



Software Architecture

When we describe the logical organization of a distributed system, there are at least the following questions we need to answer

1. What are the components that are communicating in the distributed system?
2. What communication paradigm is used?
3. What are the roles and responsibilities of each component?

Component

A component is a modular unit that uses and provides well defined interfaces and that is replaceable

Software Architectural Styles

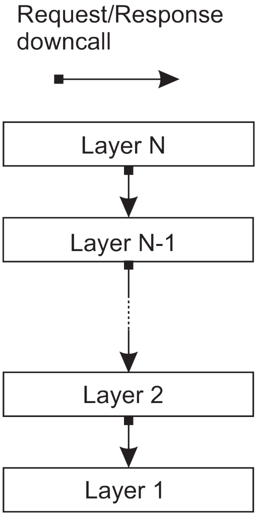
We will discuss the following software architectures

- Layered architectures
- Object-based architectures
- Service-oriented architectures
- Resource-centered architectures
- Publish-subscribe/Event-based architectures

Note: in real systems many styles can be combined together

Layered Architecture

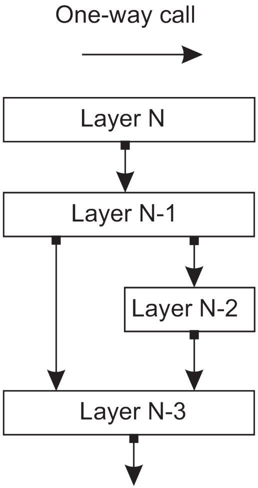
- Components are organized into **layers**
- A component at level j can make a **down call** to component at a **lower level** i and waits for a response
- A component at level i can make a **up call** to component at a **higher level** j to notify the occurrence of an event



Communication networks

Layered Architecture

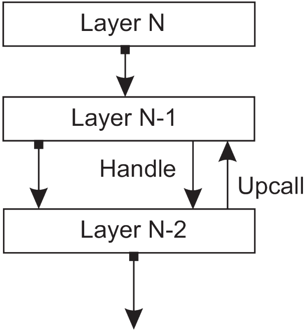
- Components are organized into **layers**
- A component at level j can make a **down call** to component at a **lower level** i and waits for a response
- A component at level i can make a **up call** to component at a **higher level** j to notify the occurrence of an event



An application can invoke a system call and a library

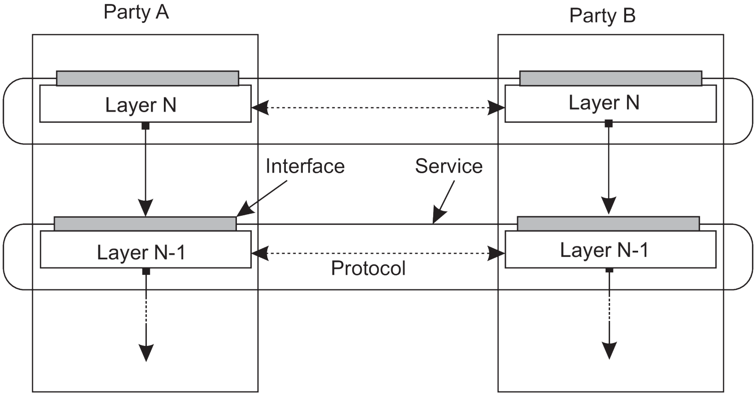
Layered Architecture

- Components are organized into **layers**
- A component at level j can make a **down call** to component at a **lower level** i and waits for a response
- A component at level i can make a **up call** to component at a **higher level** j to notify the occurrence of an event



An application registers a callback that is called when an event occurs

Example: Communication Protocols



- Each layer implements one **communication service**
- Each layer offers an **interface** that allows using the provided service
  - The interface **hides** the implementation of the service
  - It offers a **API** to set up a connection, send/receive messages and shutdown a connection

Example: Communication Protocols

TCP/IP model	IoT protocols
Application	HTTPS, XMPP, CoAP, MQTT, AMQP
Transport	UDP, TCP
Internet	IPv6, 6LoWPAN, RPL
Network access & physical	IEEE 802.15.4 Wifi (802.11 a/b/g/n) Ethernet (802.3) GSM, CDMA, LTE

- Each layer implements one **communication service**
- Each layer offers an **interface** that allows using the provided service
  - The interface **hides** the implementation of the service
  - It offers a **API** to set up a connection, send/receive messages and shutdown a connection

Example: A Simple Server in Python

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("", 8888))

while True:
  conn, addr = s.accept()
  print 'Connected by', addr
  while True:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
  conn.close()
```

The Socket Interface

`socket()`	create a socket for the connection
`accept()`	listen for incoming connections
`connect()`	set up a connection
`close()`	shutdown a connection
`send()/recv()`	send/receive data over a connection

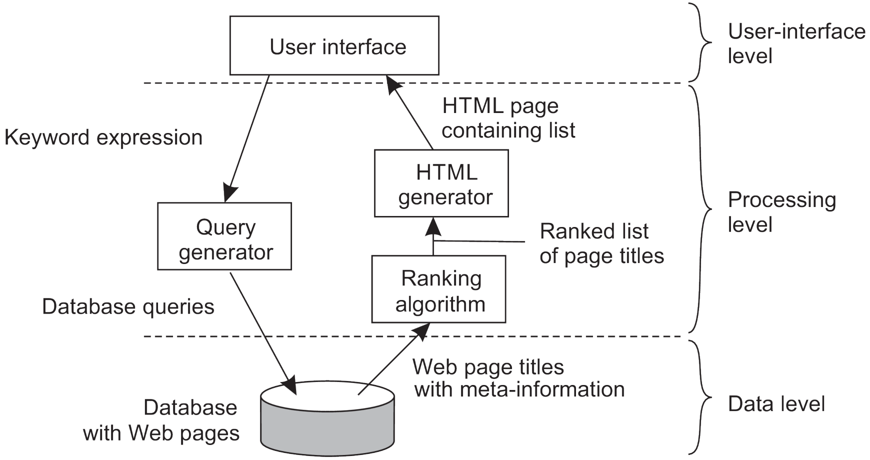
Example: Compare with the Java Socket API

Example: Application Layering

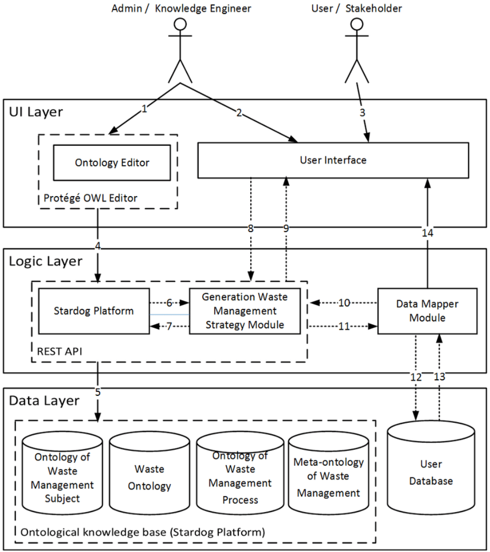
Many applications can be constructed by three different components

1. **Presentation layer:** handle the interaction with users or external applications
2. **Processing layer:** coordinate the application, process command and perform calculation
3. **Data layer:** handle the storage of data by using a database or a file system and keep them coherent

Example: A Simplified Search Engine

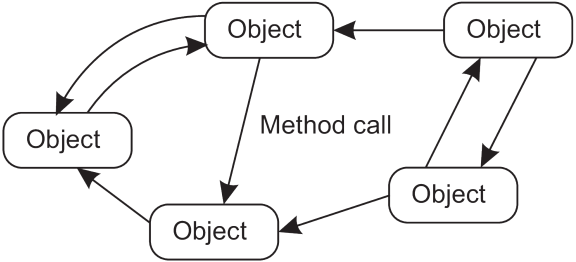


Example: A Support Decision System for Waste Management



Object-based Architecture

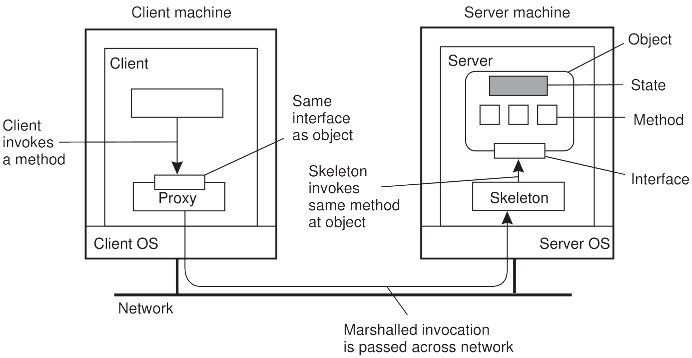
- Components are **objects** interacting through **method calls**
- Objects are on **different machines**, thus, method calls execute across the network
- Objects encapsulate the state and offer an **interface** concealing the implementation
- An object should be replaceable with another one with the same interface
- The idea is that a distributed application is composed of objects as in OOP



Note: usually, an object state is not distributed

Object-based Architecture: The Idea

- The actual object is on the server machine
- The **proxy** implements the object interface and stays on the client
- It **marshals** method invocations into messages and **unmarshals** the reply messages into results
- The **skeleton** receives invocation requests, unmarshals them, performs the actual invocations and send the results back



Example: Distributed Objects in Java (RMI)

The interface

A simple service that returns the today's date as a string

```
public interface RCalendar extends Remote {
    String today() throws RemoteException;
}
```

`RemoteException` signals that the remote invocation failed

Example: Distributed Objects in Java (RMI)

On the server

The actual implementation of the object hosted by the server

```
public class ServerCalendar extends UnicastRemoteObject implements RCalendar {
// ....

    public String today() throws RemoteException {
        Date now = new Date();
        return now.toString();
    }
}
```

`UnicastRemoteObject` signals that our object is remote, thus, stub and skeleton are needed

Example: Distributed Objects in Java (RMI)

On the server

The server publishes the object in a registry with the name “mycalendar”

```
public class Server{
    // ...
    Registry reg = LocateRegistry.createRegistry(2018);
    ServerCalendar cal = new ServerCalendar();
    reg.rebind("mycalendar", cal);
    // ...
}
```

Example: Distributed Objects in Java (RMI)

On the client

The client locates “mycalendar” in the registry and retrieves it from the server

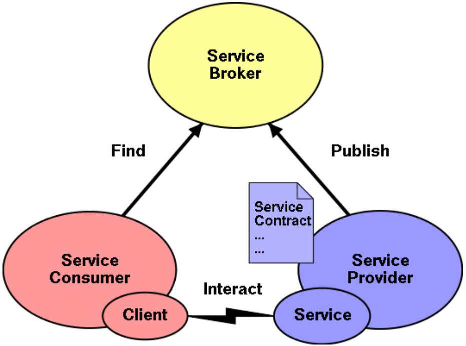
```
public class Client {
// ...
    Registry reg = LocateRegistry.getRegistry("localhost", 2018);
    RCalendar cal = (RCalendar)reg.lookup("mycalendar");
    System.out.println("Now: " + cal.today());
// ...
}
```

Service-oriented Architecture (SOA)

- Components are **services** interacting through communication protocol over the network
- A service
  - represents a specific activity
  - is black-box (encapsulation)
  - is self-contained
  - may be made of other services
  - provides a well defined interface
  - is reusable
- Services maybe implemented by different providers (different administrative organization) using different underlying technologies
- A distributed application can be thought as a service composition where services operate in harmony

Main Entities in SOA

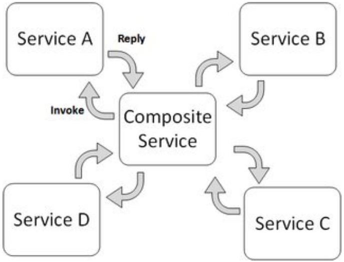
- The **Broker** makes information about services available to consumers
- The **Provider** makes available a service and provides the Broker with the required information
- The **Consumer** locates services in the Broker registry, and then bind to the service Provider to invoke it



Note: the interaction between the Consumer and the Provider is governed by the service contract

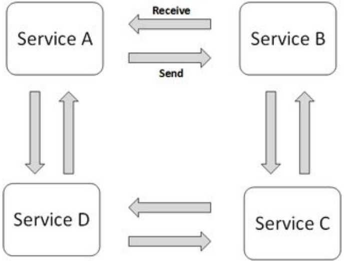
Service Composition: Orchestration

- In the system there is a special single centralized process (orchestrator) that is in charge of coordinating the interaction among different services
- The orchestrator is responsible for invoking and combining the services

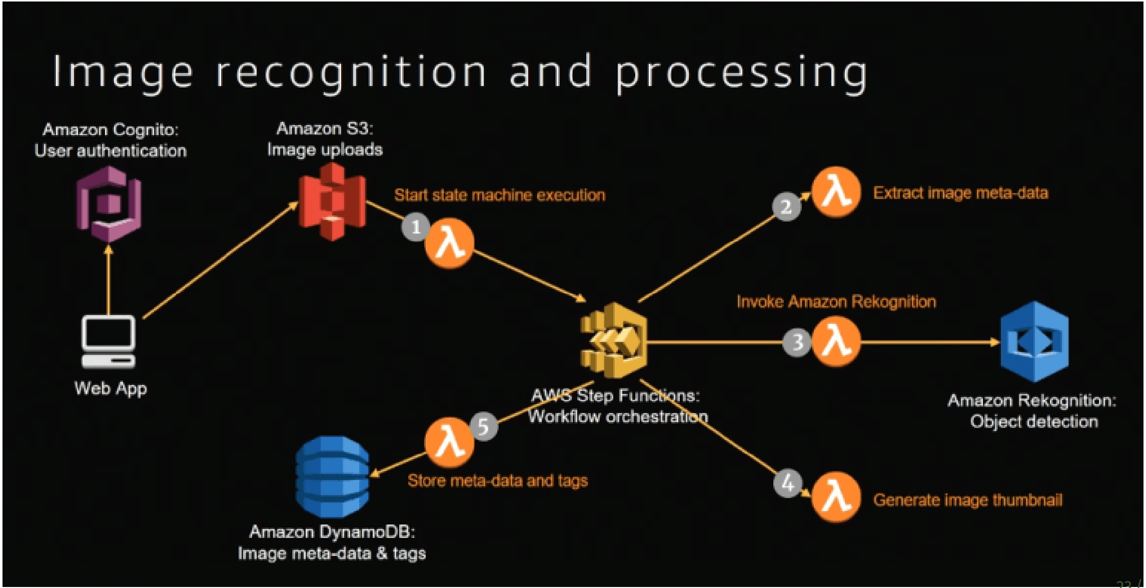


Service Composition: Choreography

- Service choreography is a form of service composition in which the interaction protocol between several partners is defined from a global perspective
- The choreography describes the interactions between multiple services, where as orchestration represents control from one party's perspective



Example: Image Recognition Engine



RESTFul Architectures

The Idea

- A distributed system is a huge collection of resources, managed by components
- Resources can be added, removed, retrieved and updated by remote components

Four key features:

1. Resources are identified by URIs
2. All services offer the same interface made of 4 operations (see next)
3. Stateless execution, no caller context being stored on the service between two consecutive requests
4. Messages are fully self-described, i.e., each message includes enough information to describe how to process the message content

RESTFul Resources

- Any information can be considered a resource: a document, a image, a temporal service, etc.
- The actual representation of the resource may be different from the representation sent to the caller
- A resource can be a singleton or a collection

`https://api.example.com/resources/`	collections
`https://api.example.com/resources/item17`	a single element

- A resource may contain sub-resources (hierarchical organization)
- There is no standard way for resource naming, but some best practices, e.g., not use query parameters to identify a resource but to provide parameters to an operation `https://api.example.com/resources/item17?format=json`

RESTFul Operations

- Typically RESTful services use HTTP as transport protocol
- REST operations are the HTTP verbs

Operation	Description
`PUT`	Create a new resource
`GET`	Retrieve the state of a resource
`DELETE`	Delete a resource
`POST`	Modify a resource

Example of call

```
GET /myapp/myresource HTTP/1.1
```

Intermezzo: observation

In the architecture we just discussed components are referentially and temporally coupled

- Referentially coupled means that during the communication components use an explicit reference to the communication partner (e.g., an address, a name, etc.)
- Temporally coupled means that in order to communicate both components must be up and running

- Pros: The interaction model is clear and relatively easy to implement
- Cons: In a high dynamic system (many processes join and leaves) this model presents some limitations (can you guess which?)

Intermezzo: exercise

What are the references in the software architecture we've discussed so far?

- Layered architecture?
- Object-based?
- SOA?
- REST?

Intermezzo: observation

The next software architecture we discuss are

- Referentially decoupled: components do not need to know each other to communicate
- Temporally decoupled: components can communicate even when one of them is not running

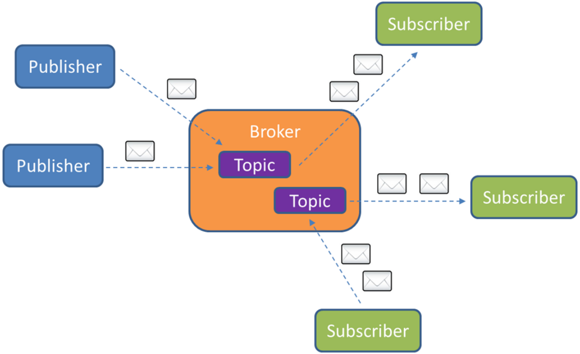
Publish-Subscribe Architecture

There are three kinds of components

Publishers broadcast messages (called events) with no knowledge of the receivers

Subscribers listen for messages (events) of interest without any knowledge of the publishers

Brokers transfer messages from publishers to subscribers



Publish-Subscribe Architecture

- Publishers & subscribers do not directly communicate but they are decoupled, the interaction occurs only through the brokers
- Brokers route the events to the interested subscribers
- An event produced by a publishers can be received by many subscribers (one-to-many communication)

Note 1: the system is open in the sense that theoretically there is no bound to the number of publishers/subscribers that can join

Note 2: brokers can be the bottleneck for event flow when the number of publishers/subscribers increase

Subscription Filters

- Subscribers express their interests in an event or a pattern of events in the form of subscription filters
- Filters are installed/removed in the system through suitable primitives (they depend on the concrete system)
- There are three methods of filtering events

1. **Topic-based:** topics work as logical channels

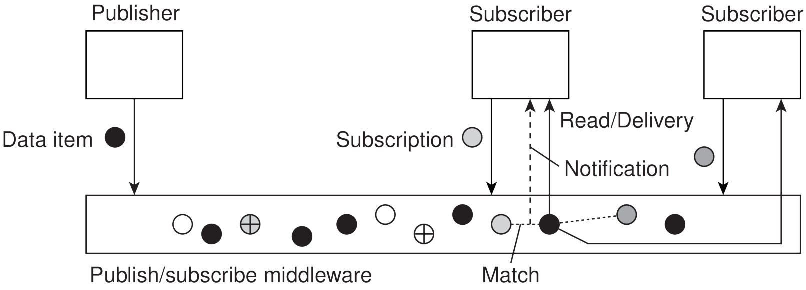
   `myhome/groundfloor/livingroom/temperature`

2. **Content-based:** all data published is structured, thus, filters are constraints/predicates expressed on attributes

   `name=Acme* and value>20$`

3. **Type-based:** events are objects belonging to a specific type, which can encapsulate attributes as well as methods, thus, subscription filter on the type, e.g., any sub-type of Exception

Dynamics



- An event corresponds to new data available
- When an event occurs the system may decide to
  - forward the published notification together with the data to interested subscribers (the system does not store data)
  - forward only the notification (the system does not store data)

Example: MQTT Protocol in Java

Publishers

```
MqttClient client = new MqttClient("tcp://localhost:2018",
                                    MqttClient.generateClientId());
client.connect();
MqttMessage message = new MqttMessage();
message.setPayload((new Date()).toString().getBytes());
client.publish("today", message);
client.disconnect();
```

- The publisher connects to the broker at port 2018
- It prepares the message and sends it with topic “today”

Example: MQTT Protocol in Java

Subscriber's callback

```
public class SimpleMqttCallBack implements MqttCallback {
  // ...

  public void messageArrived(String s,
                             MqttMessage mqttMessage) throws Exception {
    System.out.println("Message received:\n\t"+
                       new String(mqttMessage.getPayload()) );
  }

  // ...
}
```

The subscriber creates a callback class whose methods are invoked when a message arrives connects to the broker at port 2018

Example: MQTT Protocol in Java

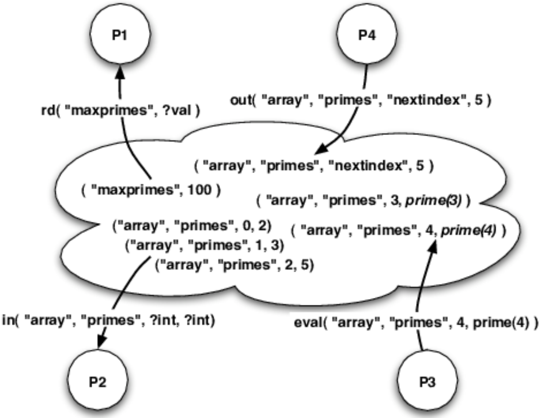
Subscriber

```
MqttClient client=new MqttClient("tcp://localhost:1883",
                                 MqttClient.generateClientId());
client.setCallback( new SimpleMqttCallBack() );
client.connect();
client.subscribe("today");
```

The subscriber connects to the broker, set the callback and registers to the topic “today”

Tuple Space

- Components communicate entirely through **tuples**
- There exists a shared data space that store tuples (tuple space)



Generative Communications

- Components put tuples into the shared data space
- To retrieve a tuple, a component provides a search pattern that is matched against tuples: all tuples matching the search criteria are returned

```
in(array, primes, ?int, ?int)
```

- The shared data space is persistent: a tuple stays there until explicitly removed/retrieved by a component
- Tuple producers and consumers do not need to exist at the same time

Integration with events

- A component subscribes to tuples satisfying a search pattern
- When a tuple is put in the data space, matching subscribers are notified

Example: Linda

Three operations

`in(t)`	remove a tuple that matches the template `t`
`rd(t)`	obtain a copy of the tuple that matches the template `t`
`out(t)`	add the tuple `t` to the data space

- The tuple space is a multi-set of tuples: a tuple t can be stored multiple times
- Both `in(t)` and `rd(t)` are blocking operations: the callers waits until a matching tuple is made available

Example: A Micro Blog in Linda

Bob

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
blog._out(("bob","distsys","I am studying chap 2"))
blog._out(("bob","distsys","The linda example’s pretty simple"))
blog._out(("bob","gtcn","Cool book!"))
```

Example: A Micro Blog in Linda

Alice

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
blog._out(("alice","gtcn","This graph theory stuff is not easy"))
blog._out(("alice","distsys","I like systems more than graphs"))
```

Example: A Micro Blog in Linda

Chuck

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
t1 = blog._rd(("bob","distsys",str))
t2 = blog._rd(("alice","gtcn",str))
t3 = blog._rd(("bob","gtcn",str))
```

System Architectures

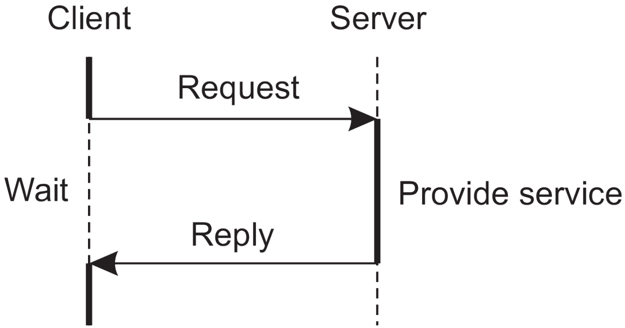
How distributed systems are organized, i.e., where software components are placed

We will discuss two organizations:

1. Centralized architectures
2. Decentralized (or p2p) architectures

Simple Client-Server Architecture

- There are components offering services (**servers**)
- There are components using services (**clients**)
- Typically, clients and servers are on different machines
- The interaction between clients and servers follow a **request-reply model**



Example: Network File System (NFS)

File system

The structure and rules used to manage and store collections of files, directories and their attributes. Typically, there are three layers

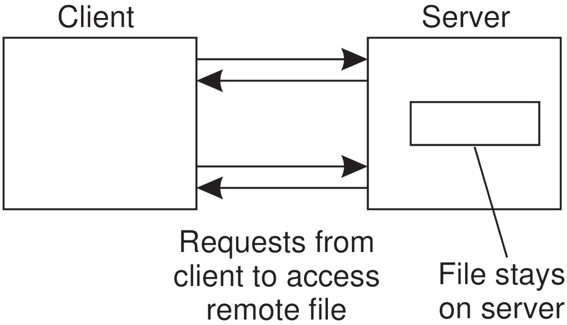
1. **logical file system** provides the API for file operations and passes the requested operations to the layer below
2. **virtual file system** provides an abstraction level that allows supporting different physical file systems concurrently
3. **physical file system** provides the physical operations interacting with the storage device

- NFS allows a collection of distributed processes to share a **common** file system
- Clients are unaware of the actual location of file (**location transparency**) and are offered an interface similar to the one of a conventional file system

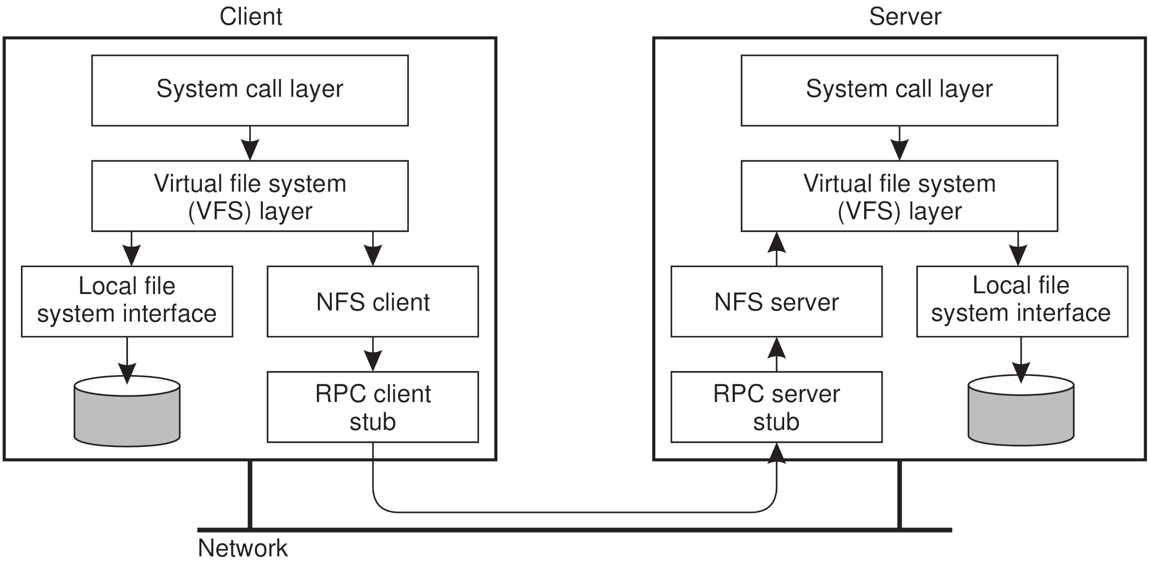
NFS Access Model

NFS implements the remote access model for access files

- Client are offered a transparent access to a file system managed by the server
- The various file operations offered to clients actually are implemented on the server
- Every operation on a file requires a communication with the server



Accessing Remote Files



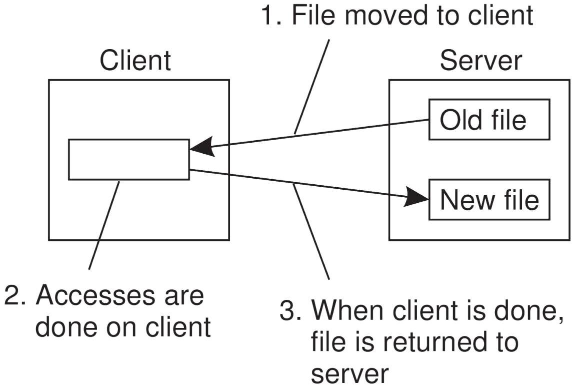
Accessing Remote Files

1. A client accesses the file system using a system call
2. The system call is managed by the VFS that dispatches the required operation to the local file system or to a NFS client
3. The NFS client forward (by means of RPC) the operation to the Server
4. In the server the request is a handled by the local VFS that interacts with the local file system

Upload/Download Model

There is an alternative approach to support operation on files that is supported by some file sharing applications (Which ones?)

- The client download a file locally, and it accesses it
- When the client is finished with the file, it is uploaded back to the server again



Recall Application Layering

An application made of three different components

1. Presentation layer
2. Processing layer
3. Data layer

How are these layers placed on machines?

Some Typical Organizations

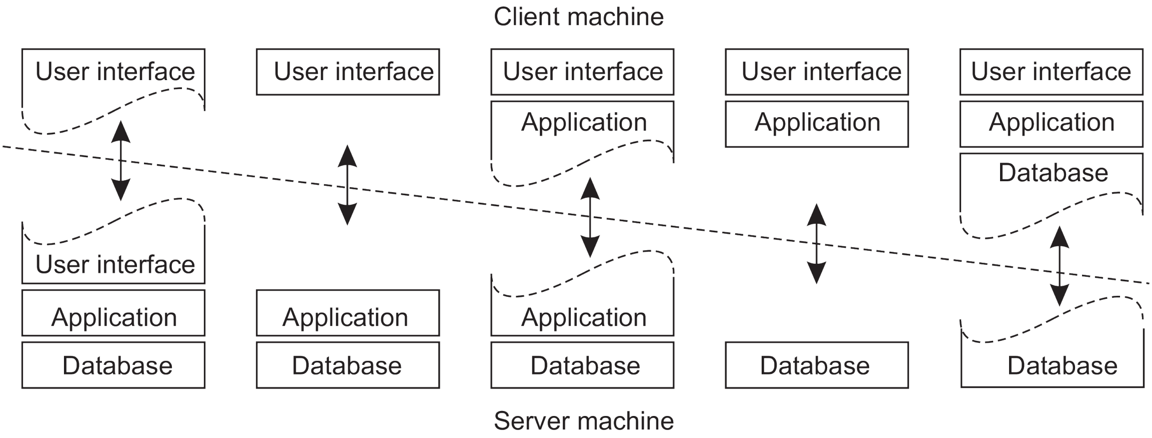
Single-tier dumb terminal/mainframe configuration, the application is on the server, the terminal allows only accessing the mainframe

Two-tier (a part of) the first layer is on the client, the other ones on the server

Three-tier Each layer on a different machine

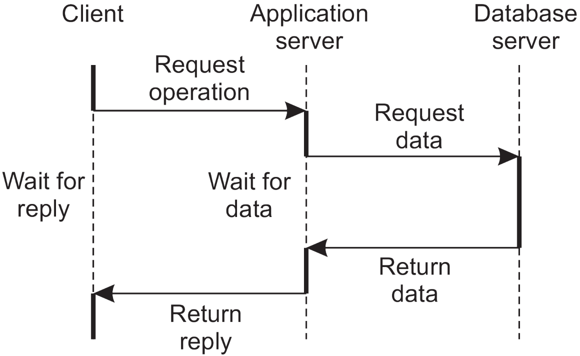
Client-server Organization in a Two-tier Architecture

Different organizations



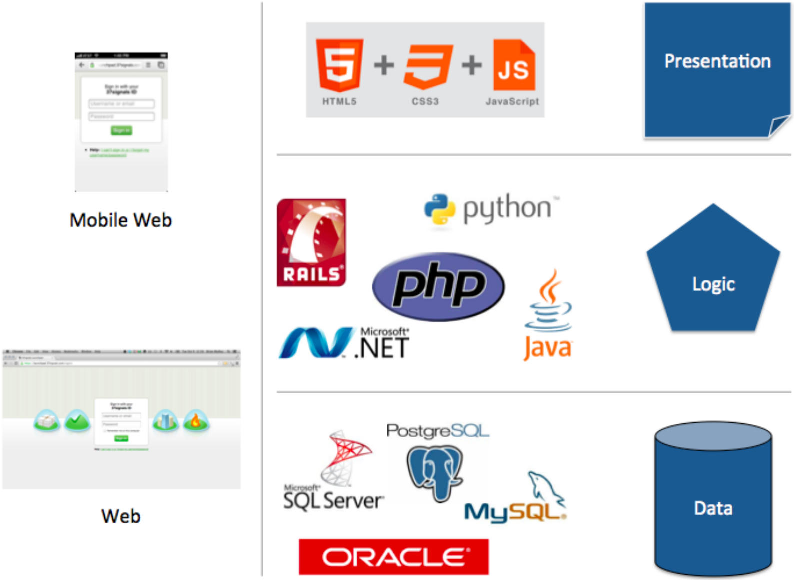
Three-tier Architecture

Each layer is placed on a different machine



Usually it is adopted by Web site: browser, web server, DBMS

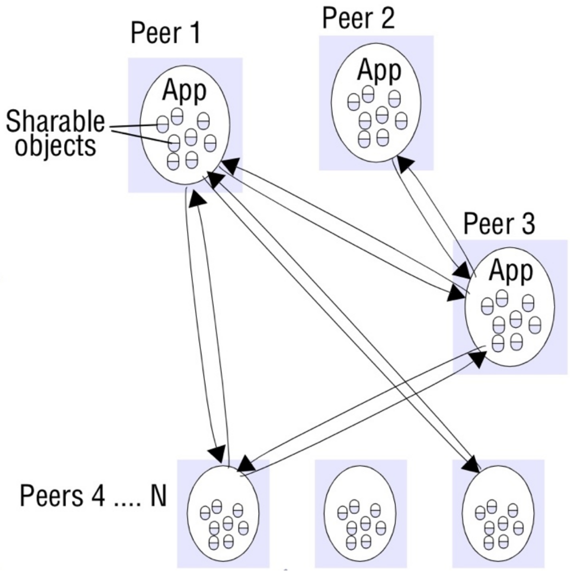
Modern Web Site Architecture



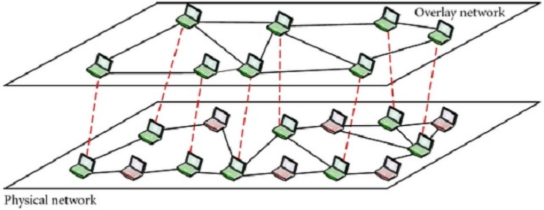
Peer-to-peer Architecture

- There is no distinction between clients and servers
- Each participant contributes to the system by sharing its resources
- The goal is to share and exploit the resources of a large number of participant for the fulfillment of a given task
- Components involved in a specific activity play similar roles, interacting **cooperatively** as **peers**: in practice, run the same program and provide the same interface
- The interaction among participants is symmetric

Peer-to-peer Architecture



Overlay Network



- An overlay network is built upon another network (e.g. the physical one)
- An edge in an overlay network may correspond a a path in the underlying network
- A overlay network is usually connected
- Each node in the network communicates only with its neighbors
- The set of neighbors can be static or dynamic

Kinds Of Overlay Network

There are three kinds of overlay networks

1. Unstructured
2. Structured
3. Hierarchical

The way how resources are denoted, located and accessed in a p2p system depends on the kind of the overlay network

Unstructured P2P Overlay Network

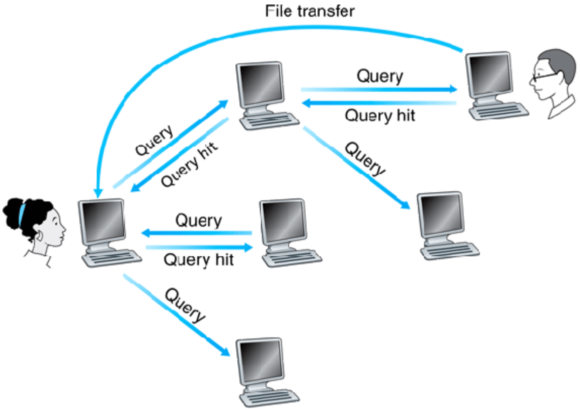
- Each peer maintains a local list of known neighbors
- When a peer joins the network, first it contacts a well-known peer to obtain a starting list of peers
- The peer selects a certain number of peers from this list
- During the execution a peer can update its local list, e.g., when a neighbor is no longer responsive
- The topology of the resulting network resembles a random graph, i.e., a graph in which a edge $\langle u, m \rangle$ exists only with a certain probability

Looking Up A Resource

- Resources are distributed across the peers participating in the system
- Looking up a resource means to find the peer which is responsible for it
- A peer only knows
  1. its local list of neighbors
  2. the resources it is responsible for
- In unstructured overlay networks there two strategies to locate a resource (another peer)
  1. Flooding
  2. Random walk

Query Flooding

- A peer passes the request for a given resource to all its neighbors
- When a peer P receives a query Q:
  - If P has seen the query before, Q is discarded
  - Else if P is responsible for the resource requested by Q, it can send back a hit
  - Otherwise, P forwards the query to all its own neighbors

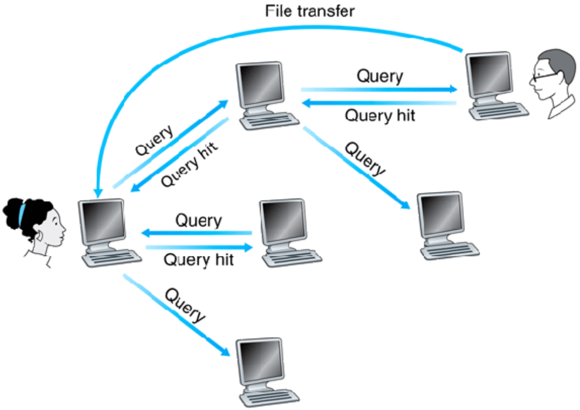


Note

- Query flooding is simple to implement
- It requires to associate a **time-to-live** TTL to each query
- It does not scale, too many queries messages exchanged in the network

Random Walk

- A peer passes the request for a given resource to a random chosen neighbor P
- When a peer P receives a query Q:
  - If P is responsible for the resource requested by Q, it can send back a hit
  - Otherwise, P forwards the query to random chosen neighbor



Note

- Simple to implement
- The initiator peer can start multiple random walks simultaneously
- It requires to associate a **time-to-live** TTL to each query

Structured P2P Overlay Network

- The network has a specific and well known topology: a ring, a tree, etc.
- Each resource in the system is uniquely associated with a key, thus, the system abstractly stores $(key, value)$ pairs, i.e., Distributed Hash Table (DHT)
- Usually, this key is returned by a hash function

$$key(data) = hash(data)$$

- Similarly, to each peer is assigned an identifier computed with the same hash function used for resources: in practice, peers and resources are mapped on the same space

$$id(peer) = hash(peer)$$

- Each peer is made responsible for storing data associated with a subset of the keys

Looking Up A Resource

- The idea is to look up a resource through its key
- The system provides a $lookup$ function that maps a key to an existing peer (that responsible for that resource)

$$peer = lookup(key)$$

- The implementation of $lookup$ performs the routing of the resource request to corresponding peer
- Usually, a good $lookup$ requires $O(\log(N))$ messages to answer a request (N the number of peers)

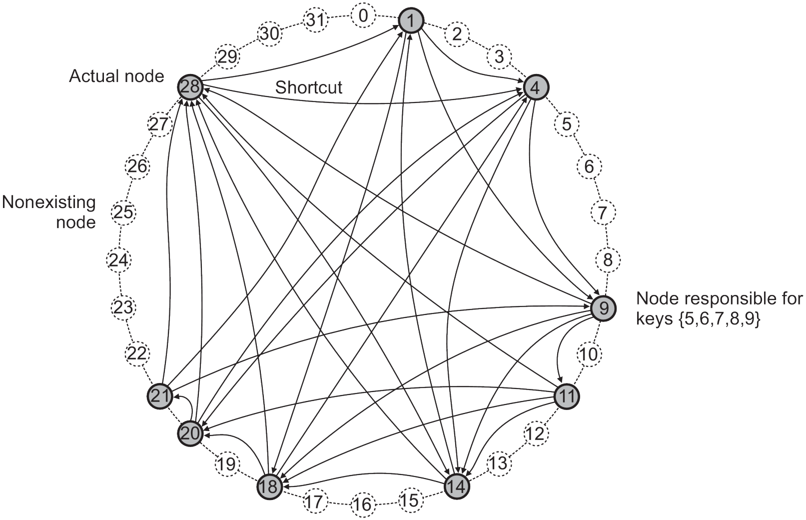
Note

There are different strategies to assign keys to peers, and to implement the research procedure used by the $lookup$

Example: Chord

- Nodes and objects are assigned m-bit identifiers that are computed using a consistent hashing function as SHA-1
- The space of keys is organized in a ring with at most 2^m elements
- An object with key k is assigned to the node with the smallest identifier $id \geq k$ (call it the successor $succ(k)$)
- Each node contains a routing table (called finger table) with m elements
- Looking up an object reduces to deliver the request to its successor

Example: Chord

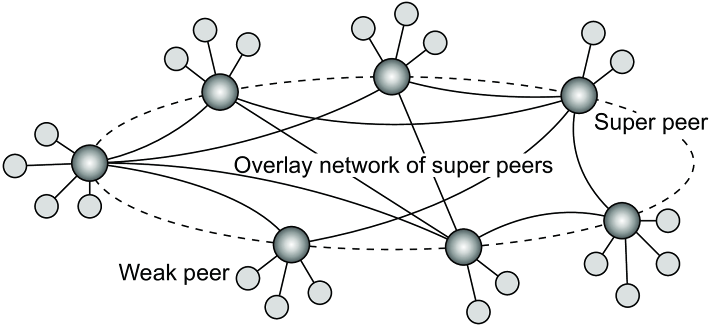


Node	Next
9	$11, 14, 18, 28$
28	$1, 4, 14$

Node 9 looks up the object with key 3:

$$9 \rightarrow 28 \rightarrow 4$$

Hierarchical Overlay Network



- Peers are classified into two groups: **super-peer** and **weak-peer**
- Every **weak-peer** is connected as a client to a **super-peer**
- All communication from and to a weak-peer passes through the corresponding super-peer
- The association between a weak-peer and its super-peer can be fixed or dynamic

Conclusion

We have discussed

- Software architecture
  - Layered organization
  - Object-oriented
  - SOA
  - REST
  - Publish-subscribe
  - Tuple spaces
- System architecture
  - Client-server
  - Peer-to-peer

References: Chapter 2 of "Distributed Systems by M. van Steen and A. S. Tanenbaum"