# Architecture of Distributed Systems 2011/2012

Architectural Styles

Johan Lukkien

# Goals

- Students understand:
  - what architectural styles are;
  - what makes up these styles through several examples.
- Introduction of terminology around distributed systems.

# Architectural styles (patterns)

- Remember: An architecture is *"the fundamental organization of a system embodied by*
  - *its components [jl: building blocks],*
  - *their relationships to each other [jl: connectors and interfaces, dependencies] and to the environment*
  - *and the principles guiding its design [jl: rules & constraints for building blocks and connectors] and evolution"*

- An <u>architectural style</u> is a *coherent set of design decisions* concerning the architecture
  - a combination of a typical (de)composition
  - and typical choices for connectors, components (building blocks) and behavior

  ....a *generic* solution for a *class of problems*

- We also have <u>interaction styles</u>, concerning just the *interaction* between building blocks
  - the nature of the connectors, and their organization

# Architectural style characterization

- Defined by
  - *motivation (guidelines, context)* for the application of the style
    - which extra-functional properties are achieved, and how
    - which problem class is solved
  - *vocabulary*
    - names for components(building blocks) and connectors, and for other concepts
  - *rules (constraints, responsibilities)* for components and connectors
  - generic *structure and behavior*
    - interfaces of components, and correspondent connectors
    - data distribution, protocols, control flow, data flow

  ....documented as a *profile (e.g. UML)*

- When applied, a style yields a partial architecture
  - in fact: styles classify architectures
- Within an architecture, several styles can be applied
  - and also, several alternatives in interaction style

- Styles encourage *communication, reuse, comparison (of alternatives)*

# Style examples

- Architectural
  - Dataflow
    - Batch sequential
    - Pipes & Filters
  - Independent components
    - Client-Server - REST
    - Publish & subscribe, broker
    - Peer to Peer
    - Service oriented
  - Shared memory
    - Blackboard, Whiteboard
    - databases, hypertext
    - Model-View-Controller
  - Virtual machines
    - Layering (also: independent components)
    - Interpreter
    - Mobile agents
    - Rule-based
- Interaction
  - procedure/method call
  - message passing
  - streaming
  - active messages
  - remote procedure call
  - remote method invocation
  - files
  - eventing
- Note: interaction styles induce some architectural elements
  - e.g RPC infra structure, client / server libraries, message queues

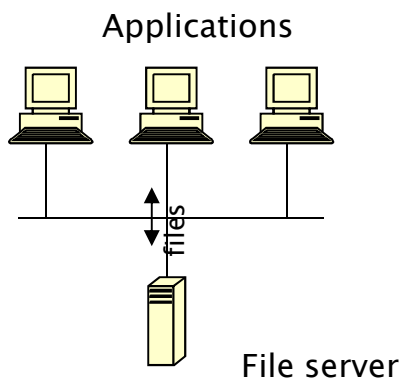
# Remember viewpoints

- While describing an architecture, different view points are taken
  - user interaction view (and other views defined by Kruchten)
  - data view: focus on how data flows through the system, and is stored
  - control view: focus on control flow
  - decomposition view

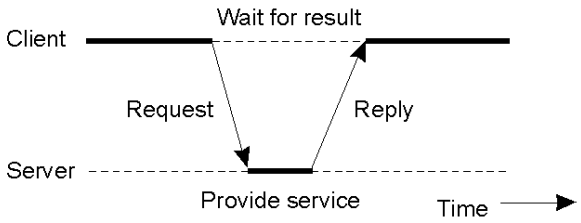
- The importance of these views often determine choices of architecture styles
  - i.e., architecture styles are best understood with respect to a view point

# Client-Server style

- Motivation:
  - sharing some localised resource (e.g. file store, compute server)
  - protecting and managing content (e.g. a database)
  - delay binding, decrease dependencies (independent development)
  - generally: separation of concerns
- Vocabulary
  - client, server (building blocks)
  - request, reply (interaction)
  - server discovery
- Rules
  - Server:
    - provides a service according to the above motivations
    - passive, awaiting requests from clients, does not know clients
    - handles data access, data integrity
  - Client
    - active, initiates activity, *discovers* server
  - no connection among clients
  - no, or limited, state on the server per client
- Structure

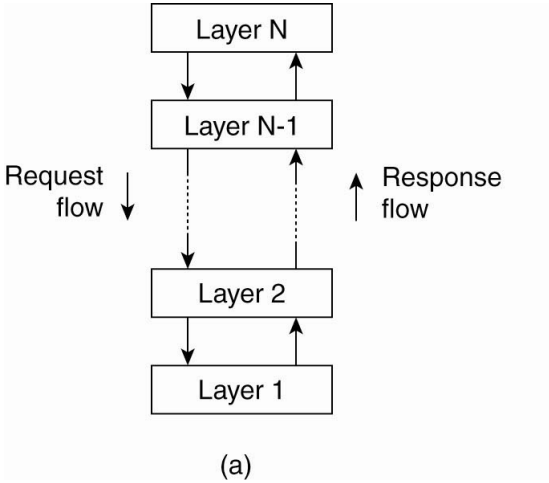



- Typical behavior
  - client finds server access point through discovery
  - regular interaction:



# Layered style

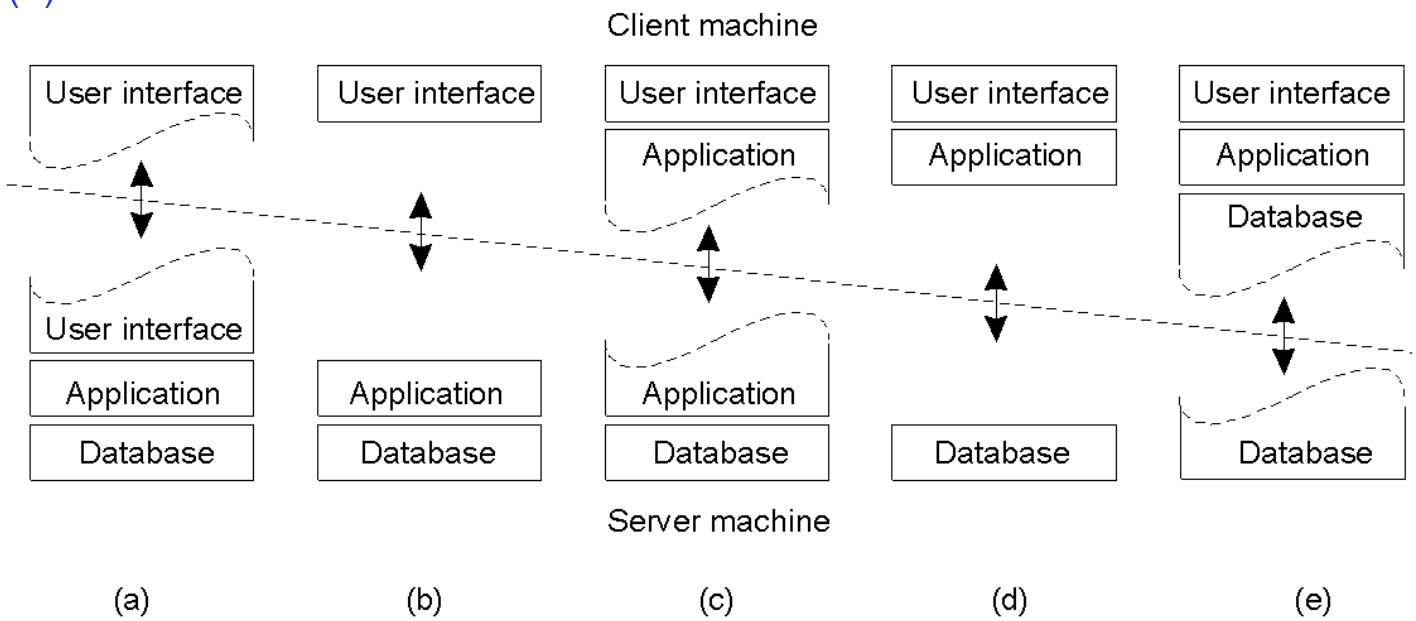
- Motivation:
  - independent layers of abstraction
    - separate concerns, limit dependencies, allow replacement
- Vocabulary
  - layer (building block), service, interface
  - call, call-back (interaction style)
- Rules
  - Layer *N* may know only layer *N-1*
    - not vice versa (why?)
    - Layer *N* is in control of binding the layer *N/N-1* interface
  - Layer *N* may not use knowledge of lower layers than *N-1*
  - Control flow can be both synchronous and asycnhronous ("event", "call-back")
- Structure



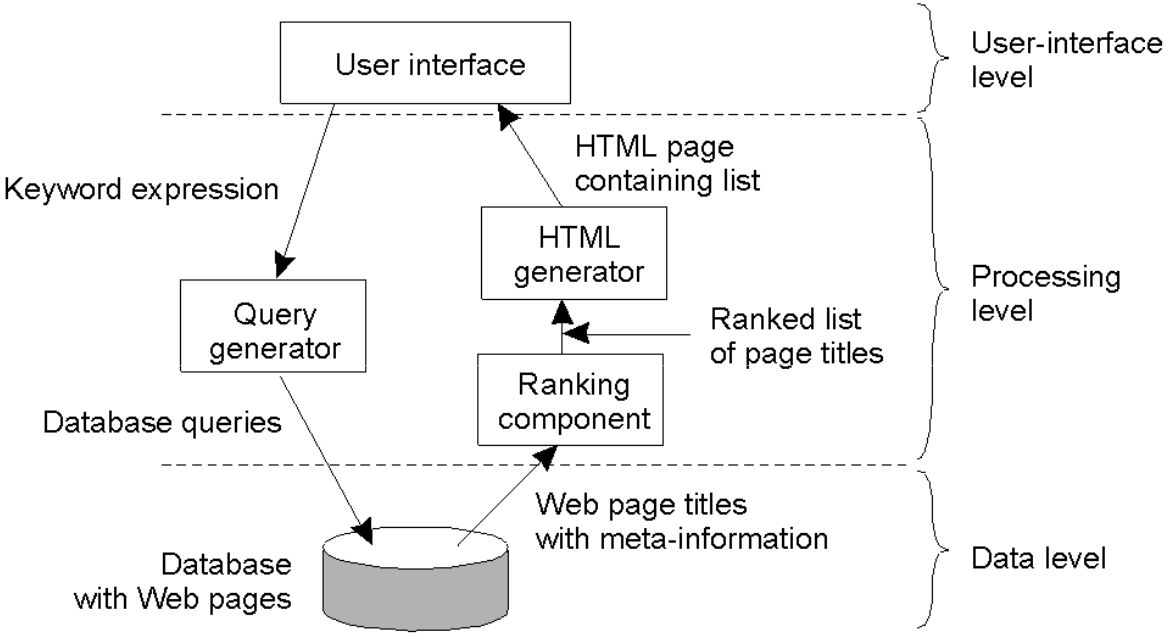

- Typical behavior
  - (Objects in) layer *N* call upon the API of (objects in) layer *N-1*
    - regard layer *N-1* as the single service provider of all lower functionality
  - Layer *N* registers *call-back* routines with layer *N-1*

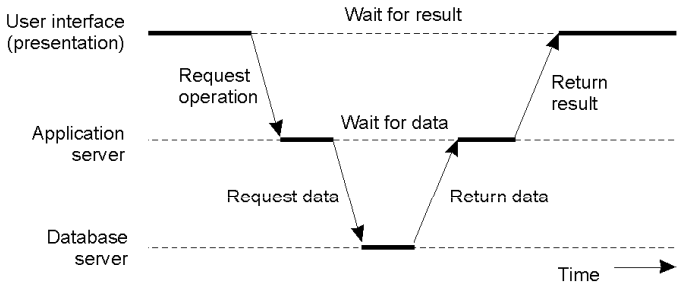
# Layers and client / server functionality

- 3-tier organization of data access: cuts possible at many places
  - (a,b) thin client – changing control
    - the user interface has small overhead; it typically runs on a machine that is a client to the application server
    - however, after connection, the user interface becomes a server for the application
  - (c,d) thick client
  - (e) file server



# Combining layering and C/S in a distributed fashion

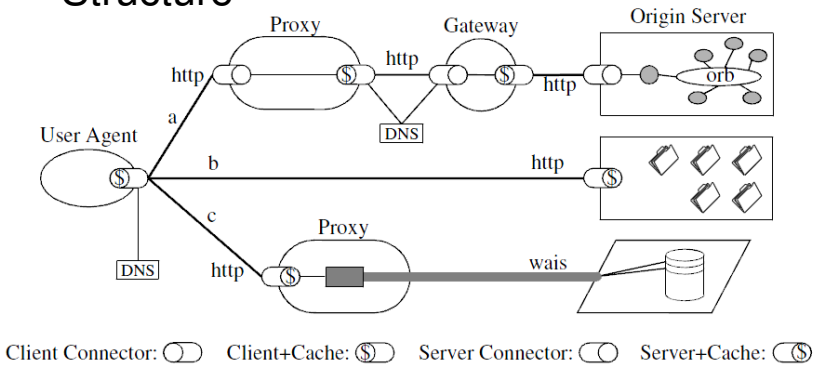



- Development (logical): layering ("presentation, business and data tier (layer)")
- Deployment: Client-Server interactions between the components
  - also called: mult-tier client-server organization
  - note: middle layer plays both client and server *roles*
    - that is actually what layering + request/reply is

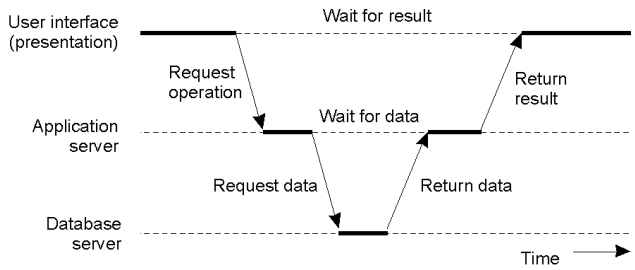
# REST: REpresentational State Transfer

- Motivation:
  - special usage of C&S aiming at
    - portability
    - independent development & deployment
    - reduction of interaction complexity
    - reliability
    - scalability
- Vocabulary
  - user agent, origin server, gateway, proxy
  - cache, layer
  - state
  - code-on-demand
  - resource, resource identifier (e.g. URL), representation, metadata
- Rules
  - stateless communication (no server state)
  - response is labeled as cache-able, and can then be cached
  - uniform interface between components
    - decouples structure from functionality
  - layering
  - client functionality can be adapted by code-on-demand
- Structure




(model in the process view from R. Fielding, Architectural Styles and the Design of Network-based Software Architectures)

- Typical behavior
  - as (layered) C&S, but only a single service (interface)




- Prime example: WWW

# Peer-to-peer style

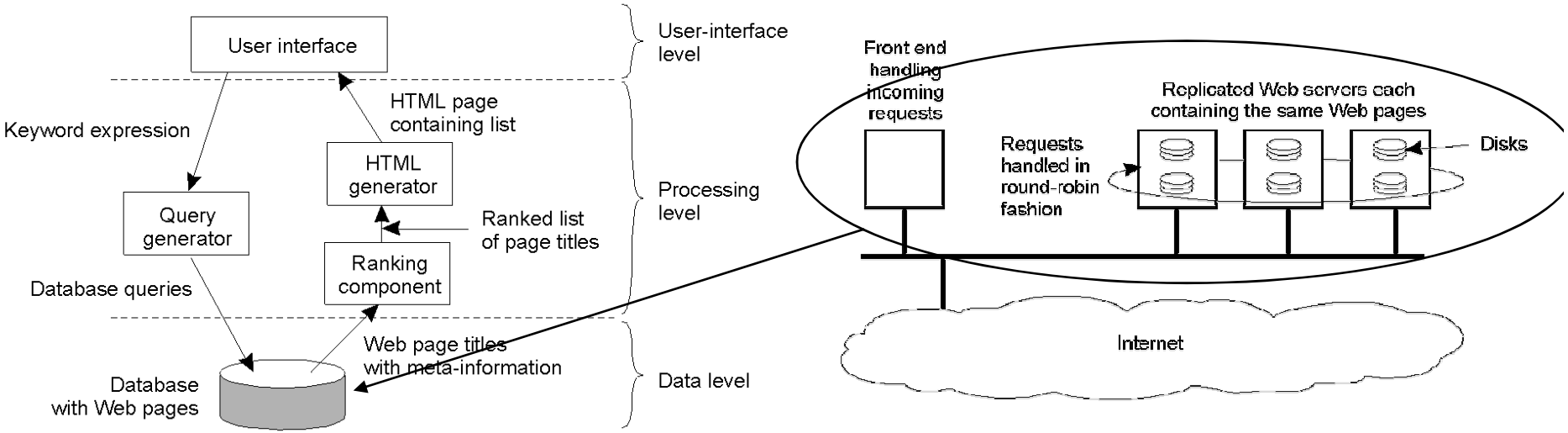
- Motivation:
  - sharing resources and content
  - cooperation in communities
  - symmetry in roles
  - increase concurrency
- Vocabulary
  - peer, super peer, distributed hash table (building blocks)
  - community (of peers)
  - overlay, application level multicast, peer discovery
- Rules
  - Peer:
    - symmetric in functionality, and contribution
    - can discover, reach and cooperate with all other peers, in principle
    - can be both passive and active
  - Super peer
    - peer that contributes extra resources to the community (extra services, like directory, discovery)
- Structure:
  - overlay, on top of network technology (from deployment view)

picture from wikipedia

- Typical behavior
  - peer finds community access point
  - joins community
  - provides passively services to other peers
  - actively uses services from peers in the community
  - the collective services of peers result in new functionality
    - e.g. file sharing

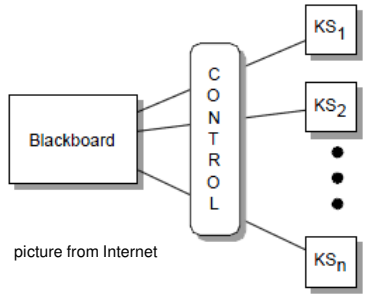
# Combine peer-to-peer with previous

- Vertical distribution: map entire *layers* to machines
- Horizontal distribution:
  - divide a *layer* across a collection of symmetric *peers* (picture)
    - e.g. a distributed database, distribution of business logic
  - divide different functions inside a layer across (asymmetric) peers



# Blackboard style

- Motivation:
  - different tasks on same state (data set )
  - current state is result of (serialized) operations of different tasks
  - tasks are coupled only through the shared state; communicating partners do not know each other
- Vocabulary
  - blackboard, knowledge sources, control component, data space
  - ('knowledge-source' derived from the envisaged domain of problem solving)
- Rules
  - Blackboard (data space):
    - contains all state
    - is passive
  - Control
    - transfers control among KS's
    - conflict resolution
  - Knowledge source
    - algorithms for state adjustment
    - generation of new data
    - don't know each other
- Structure:
  - knowledge sources access blackboard under control of *Control*



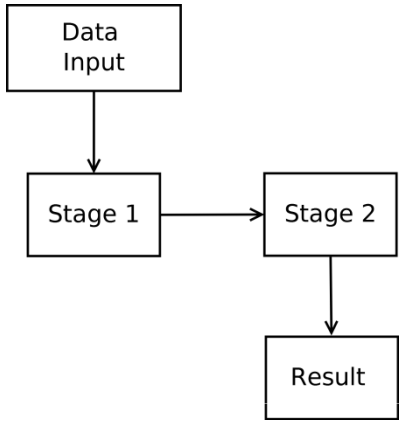

picture from Internet

- Typical behavior
  - KS's register with Control
  - Repeatedly, Control determines *enabled* KSs', selects and calls one to execute
- Metaphore
  - several people solving a problem jointly, by writing on a blackboard

# Batch sequential style

- Motivation:
  - Processing in several stages
  - Monolithic component would be too complex
  - Reconfigurable system: stages inserted/removed/reordered easily
- Vocabulary
  - Processing steps (stages), batch of data (single input set), sequential processing
- Rules
  - Every stage is stand alone in processing. There is no shared state.
  - Only subsequent stages exchange data.
  - System handles input sets one by one. No stream of data. No concurrent work of stages on different data.
- Examples
  - Compilers
- Weak point
  - Not interactive and slow due to not using concurrent processing

- Structure:

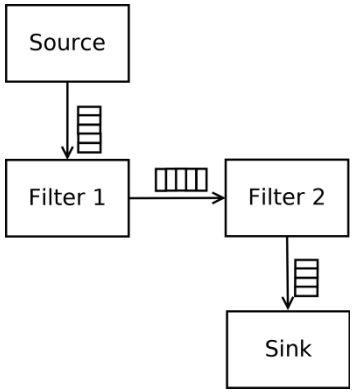



- Typical behavior
  - Stages designed to be reusable (e.g. have configurable parameters)
  - Different stages can be on different processors, but their execution is still sequential.
- Metaphore
  - Assembly line that can start processing next product only after previous one has been completely processed

# Pipes & filters style

- Motivation:
  - Systems that work with streams of data (e.g. video processing multimedia systems)
  - Streams of data are naturally processed in several stages
  - Reconfigurable system: stages inserted/removed/reordered easily
- Vocabulary
  - Pipes, filters, data stream, data source, data sink, buffers, pipeline
  - Data stream is processed by filters
  - Filters are connected via pipes (explicit connectors) .
  - Forks and joins are allowed, but pipeline is sequence of filters from data source to sink.
- Rules
  - Pipes handle communication & buffering
  - Every filter is stand alone stage in processing. There is no shared state.
  - Only adjacent filters exchange data
- Weak points
  - Difficult to share global data
  - Difficult to handle control (e.g. filter crashes)
  - Not convenient for interaction

- Structure:

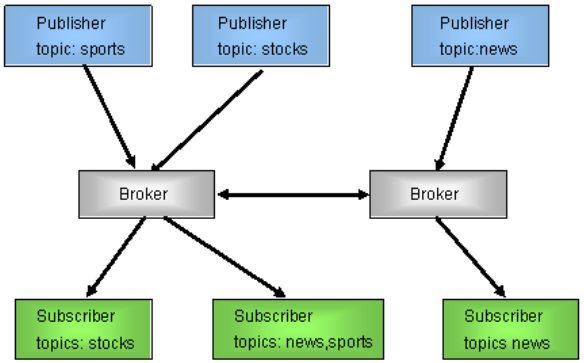



- Typical behavior
  - Filters designed to be reusable (e.g. have configurable parameters)
  - Concurrent processing by splitting the flow, but also inherent because filters concurrently work on different data
  - Different stages can be on different processors. Pipes need to provide streaming data from one processor to another

- Metaphore
  - Stream of products being processed along assembly line
  - Example: Unix shell commands, GStreamer

# Publish/subscribe style

- Motivation:
  - Decoupling data producer from data consumer
  - Sending data when it is available. Avoiding need to poll for data.
  - Allow multiple consumers
  - Allow runtime changes of set of consumers
- Vocabulary
  - Publisher, subscriber, subscription, notification, topics, broker
  - Subscriber registers with publisher to receive notifications for chosen topic.
- Rules
  - Every subscription relates a topic to 1 subscriber.
  - Publisher can have multiple subscribers for same topic.
  - Notification goes to all subscribers; however, subscribers may specify receive policies
  - When existent, broker decouples the publishers and subscribers
- Structure:



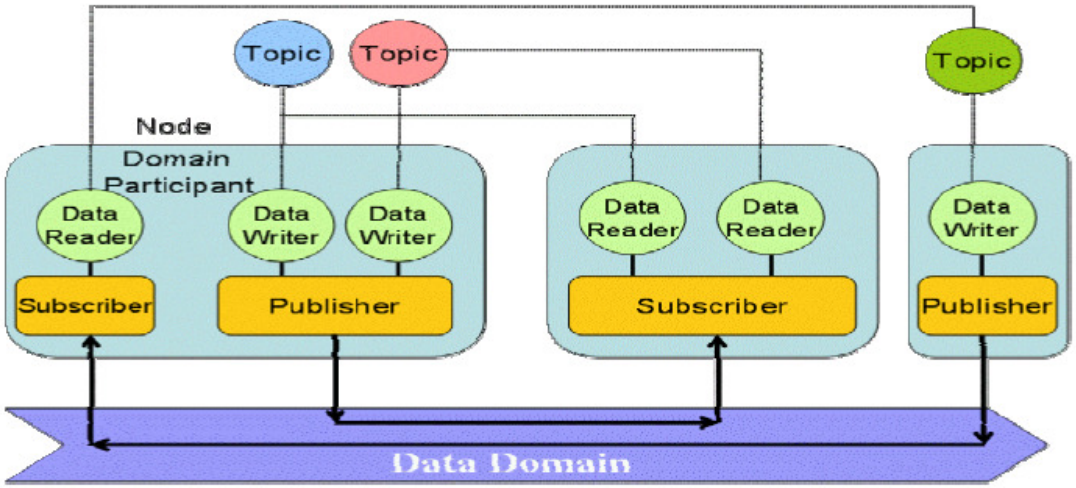

- Typical behavior
  - Subscribers find brokers and send subscriptions for topics of interest
  - Publisher register with brokers, or are discovered
  - Notification may be only about events, or it can also contain data.
  - Broker can be omitted
- Metaphore
  - Newspaper/magazine subscription
  - Observer design pattern is an example of publish/subscribe style

# Example: data distribution service (DDS)

- OMG (Open Management Group) standard
  - **Data-Centric Publish-Subscribe (DCPS)** – low level interface

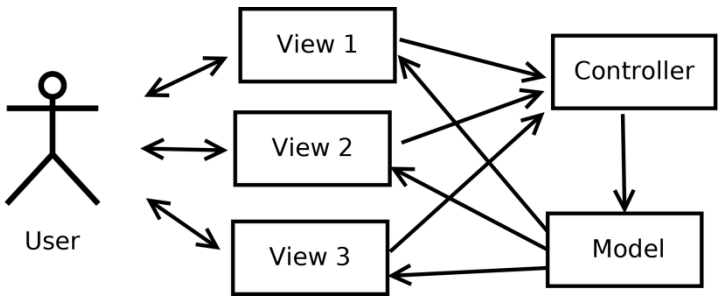
    provides a global data space. Type-specific data publishers send data that interested subscribers can receive.
  - **Data Local Reconstruction Layer (DLRL)** - optional layer built on top of DCPS

    DLRL providing more transparency by hiding much of the pub/sub details and allowing for distribution of an object model.
- Implementations
  - open source: OpenDDS (implements DCPS), OpenSplice (real-time variant)…



# Model – View - Controller style

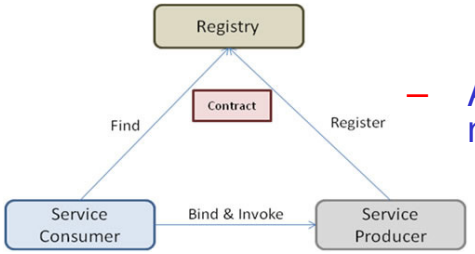
- Motivation:
  - Multiple views in interactive applications need to be kept synchronized
  - 'Look and feel' of user views tends to change a lot over time (sometimes even underlying technology is changed)
  - decoupling views from model facilitates change of views without need to rewrite model as well
- Vocabulary
  - User interacts with system via views
  - Model contains all data represented by views.
  - Controller updates the model
  - Views present model to user
- Rules
  - View does not handle user input. It forwards it to controller
  - Only controller can initiate updates of the model
  - Model does not change views. It notifies them about change
  - Views query relevant parts of model and update their representations of it
- Structure:



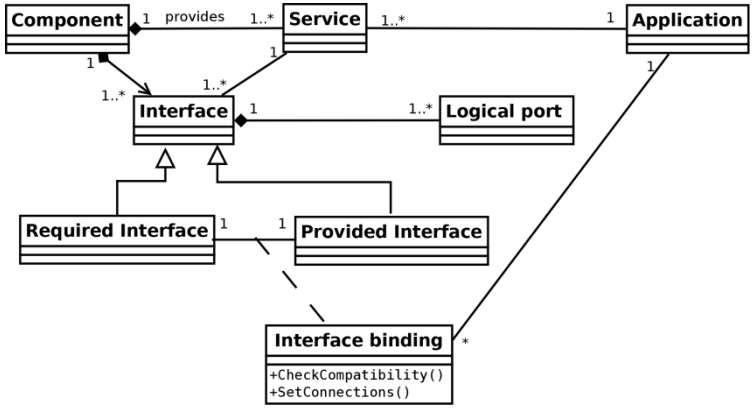

- Typical behavior
  - User interacts with a view resulting in model changes through controller
    - Possible: separate controller for every view
  - All views are updated accordingly
    - Notification mechanism can be based on publish-subscribe style
- Weak points
  - Changing model has consequences for most or all views
  - Not convenient for very frequent events
  - Too complex for simple interfaces

# Service oriented (SOA) style

- Motivation:
  - Separate *functionality* ("service"), the *implementation*, the *deployment context* and the *application context*
  - Build applications by very late (dynamic) binding
  - Integration of enterprise information systems
- Vocabulary
  - SOA, service, interface, discovery, composition, binding, orchestration, choreography
- Rules
  - Application is built ad-hoc out of services that communicate in a standardized manner
    - via a network
    - see e.g. REST
  - Service is a self-contained functionality. It does not depend on state of other services, or of the system (OS, language) it is running on.
  - Services are discoverable

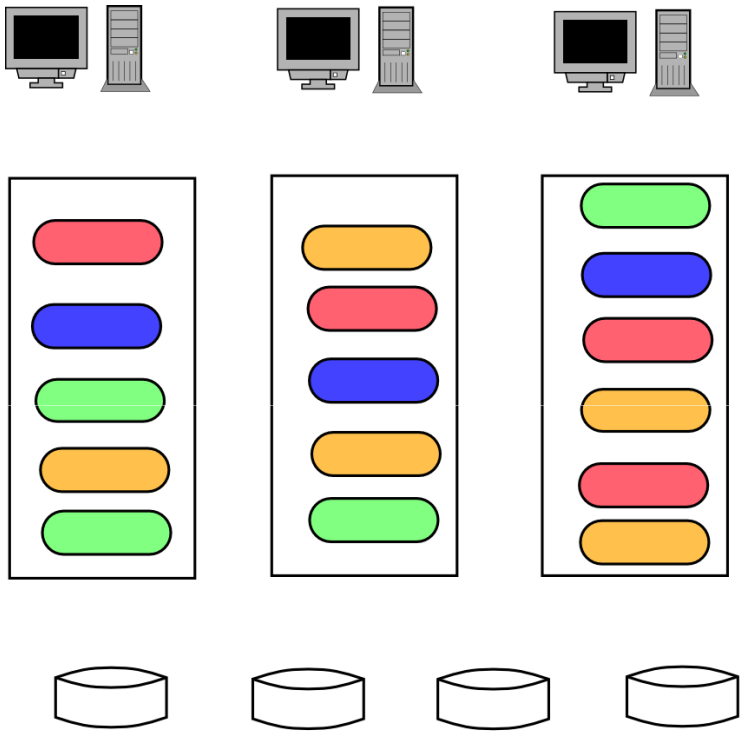



- Structure (conceptual):

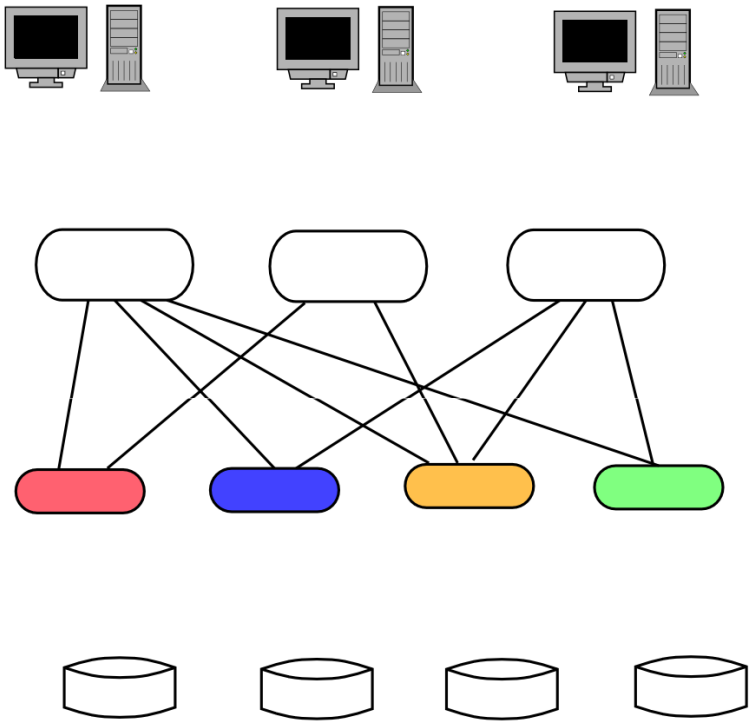



- Typical behavior
  - Providers publish services; Applications ("orchestrations") discover services, and bind their interfaces
    - Service broker (registry) can exist to manage discovery process
  - Applications send data objects through a number of services as a work flow
    - Often XML based RPC (SOAP)
    - Services are typically kept simple and focused on single task
- Note: SOA is a huge research domain in business information systems

# Enterprise application organization



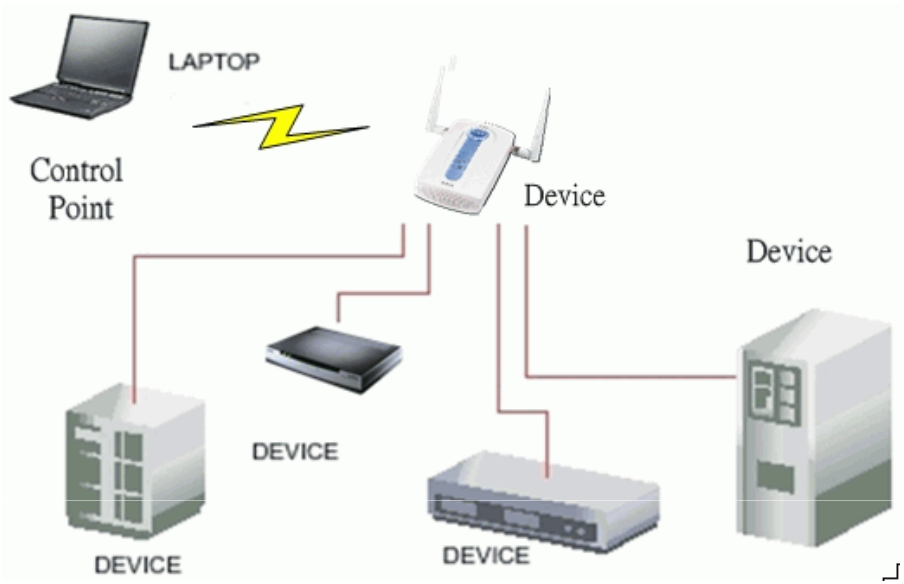
**Monolitic applications**



**Tiered service architecture**

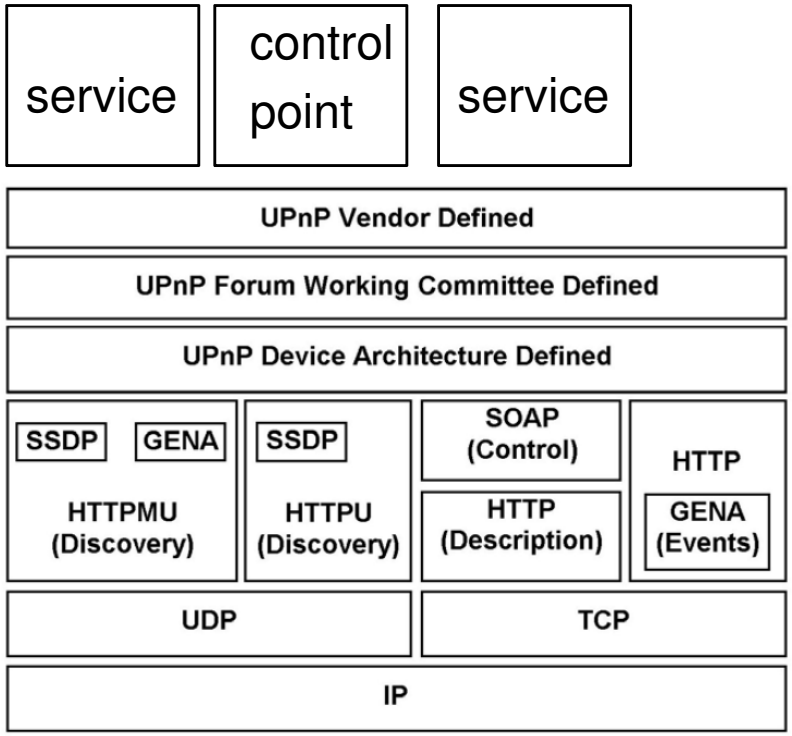
# Example: UPnP (part of DLNA)

- Universal Plug'n Play
- Two types of entities: *devices* and *control points*.
  - Devices are service providers, control points are service users.
- Discovery protocol:
  - A device added to the network can advertise its services.
  - A control point added to the network can search for devices (and then services) of interest.

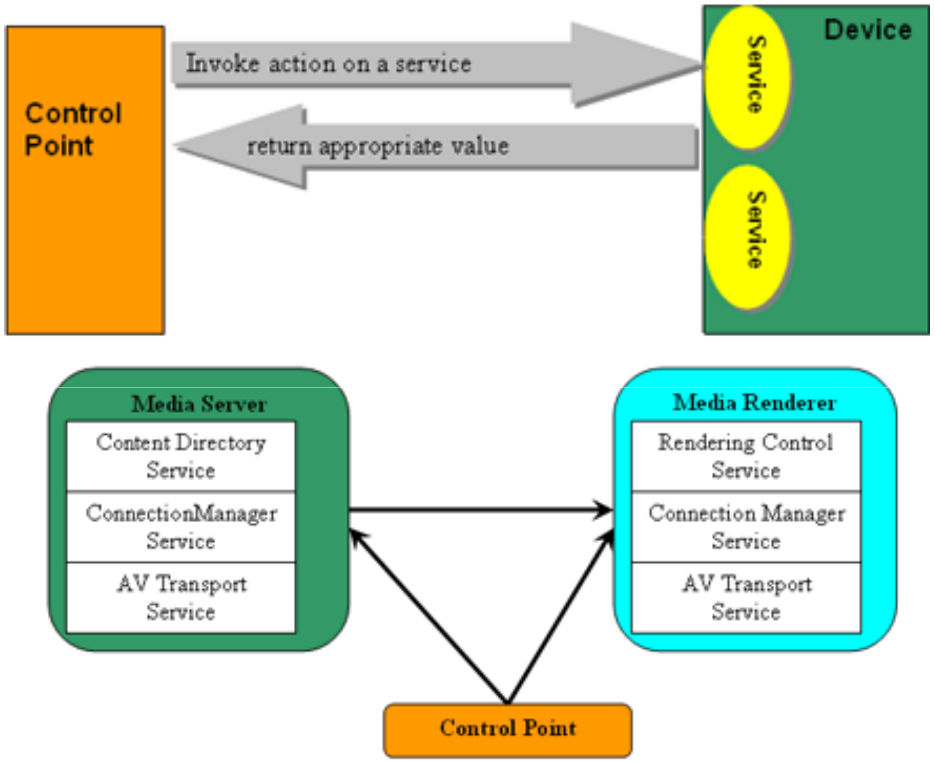



- Description:
  - A control point can obtain a detailed (XML) description of a device. Description contains information about services offered by the device and their interface
- Control points
  - can invoke actions offered by services
  - can subscribe to events published by services
  - and can access URLs with presentation provided by devices
  - can bind interfaces of services

# Logical organization, interaction




A typical UPnP application uses an API to use functions of Remote Procedure Call and service discovery. Standard protocols are used for the core protocols of the standard.

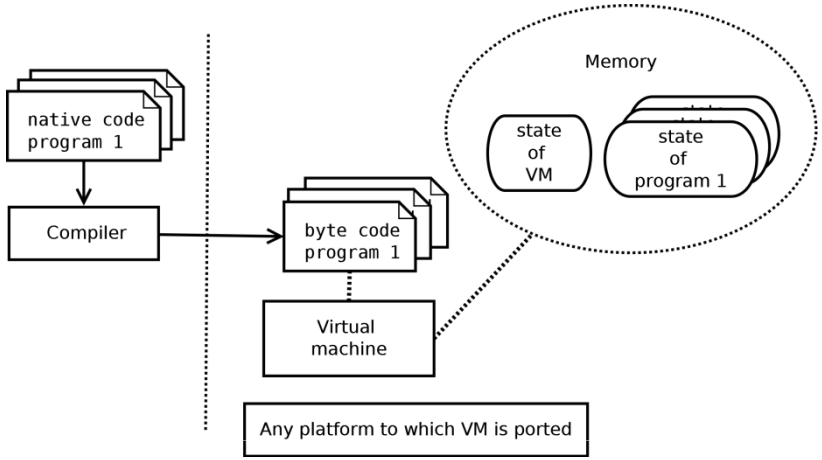



Interactions: control points call actions on services (top) or establish connections between services (bottom)

# Virtual machine style

- Motivation:
  - Portability (eliminate platform dependencies)
  - Compact task representation
  - Code mobility
- Vocabulary
  - Platform, virtual machine (VM), native code, byte code, compiler, memory, state of VM, state of program
  - Programs are written in native code. This human understandable form is compiled to byte code, interpreted by a Virtual machine.
- Rules
  - Actual execution of program(s) is inside virtual machine
  - Same byte code can run on any platform (OS, ISA). Virtual machine needs to be ported.
- Weak points
  - Performance penalty (e.g. on same machine Java programs run typically slower than C/C++ programs)

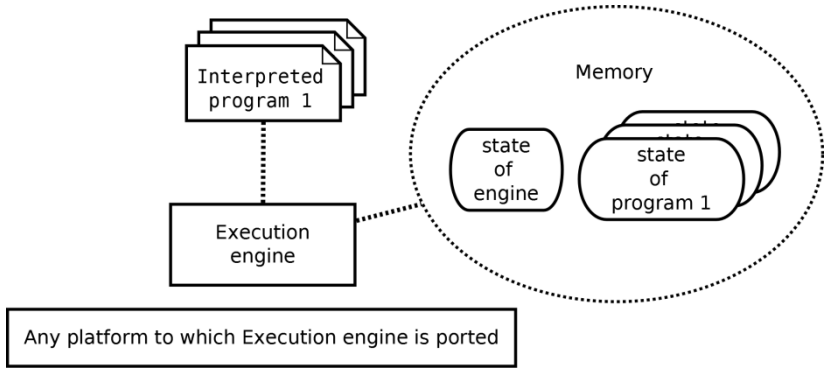
- Structure:




- Typical use
  - Compile once (locally), run anywhere
    - part of REST (mobile code)
  - virtual machine provides API for access to the local platform
- Metaphore
  - A speaker needs to hold his speech in different countries. He translates his speech into English. Speaker reads English version to audience, while translator converts this to the language of the audience.

# Interpreter style

- Motivation:
  - (Same as for VM)
  - Domain-specific language, with high-level concepts from the domain
- Vocabulary
  - Platform, interpreter (execution engine), interpreted program, pseudo-code (script), memory, state of engine, state of program
  - Programs are written in a high level pseudo code or script. Execution engine interprets this code.
  - Main difference with VM: absence of intermediate machine language
- Rules
  - Actual execution of program(s) is inside interpreter
  - Same pseudo code can run on any platform. Execution engine needs to be ported.
- Weak points
  - Performance penalty
    - larger than for Virtual machine as there is no machine language that can be optimized
- Structure:




- Typical behavior
  - Programmer writes script with high expressive power (awk, perl…)
  - Scripts are deployed to a variety of machines
- Metaphore
  - A speaker needs to hold his speech in different countries. Speaker talks in his own language to audience, while translator simultaneously converts this to the language of the audience.

# Choosing an architectural style

- No single solution: the same problem can be tackled with many different designs and architectural styles
- Choice depends often on extra-functional requirements:
  - Costs (resource usage, development effort needed)
  - Scalability (effects of scaling work complexity and available resources)
  - Performance (e.g. execution time, response time, latency..)
  - Reliability
  - Fault tolerance
  - Maintainability (extending system with new components)
  - Usability (ease of configuration and usage…)
  - Reusability

# Architecture style application

- An (*architectural) framework* is a tool that supports implementing instances of a style
  - it support the concepts, the vocabulary of that style
  - it defines (part of) the development *procedure,* it *automates* the *generation* of standard *components* and *interfaces* as well as the *application* of the *constraints* (rules)
  - it supports the interaction styles deemed suitable for the style and automates their use
    - we can also have a framework just for the interaction style, e.g. an RPC framework

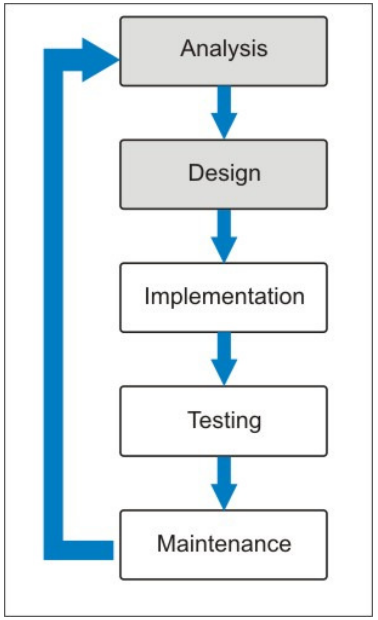
- A framework can be a method, but also a software tool

- A framework defines part of the *life cycle* of the software or system it is used for

- A framework may include a programming model

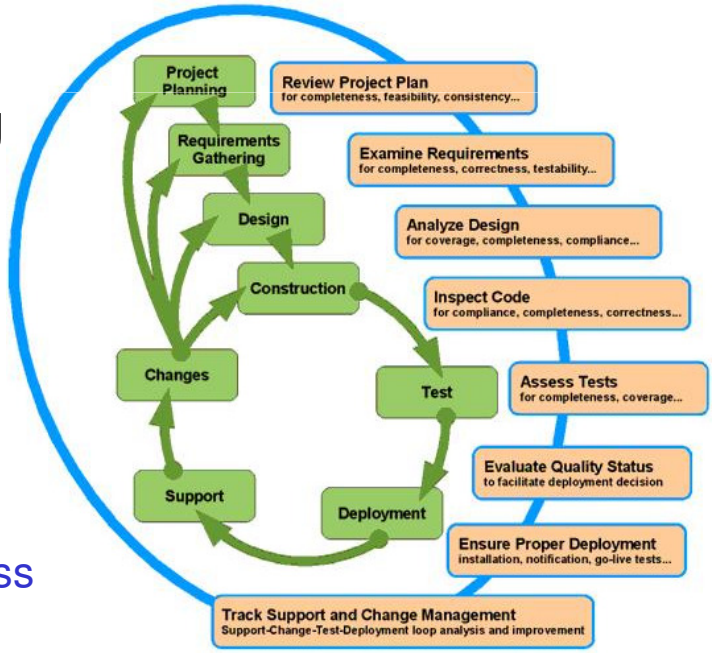
# Life cycle

- *The* life cycle *of a product or system is the series of stages it goes through from inception to decline*




- A typical life cycle for a software system is given to the right

- More detail is obtained by adding information regarding the activities in the stages




- Notes:
  - also system parts have life cycles, affecting the overall system
  - the life cycle should also address evolution, redesign

# Programming model

- A Programming Model is set of *concepts*, *primitives* and *combinators* to express a computation or system behavior

- Examples:
  - the language C: variables, functions, assignments, expressions, function calls, repetition, selection, sequential composition
  - C++ adds objects, classes, inheritance as concepts, and primitives to use them
  - An RPC programming model has concepts and primitives to manage and express an RPC-based system
  - languages like Haskell, Perl, Prolog
  - Google's MapReduce: functions *Map* and *Reduce*

# Some questions

- Can we find situations in which a particular style is *not* suited?
  - which extra-functional properties are (not) achieved by the styles?

- Apply different styles to the same problem, to obtain insight.
  - Web browsing using Blackboard, Peer-to-Peer?
  - A distributed file system?
  - A concurrent computation?
  - A bus control system?

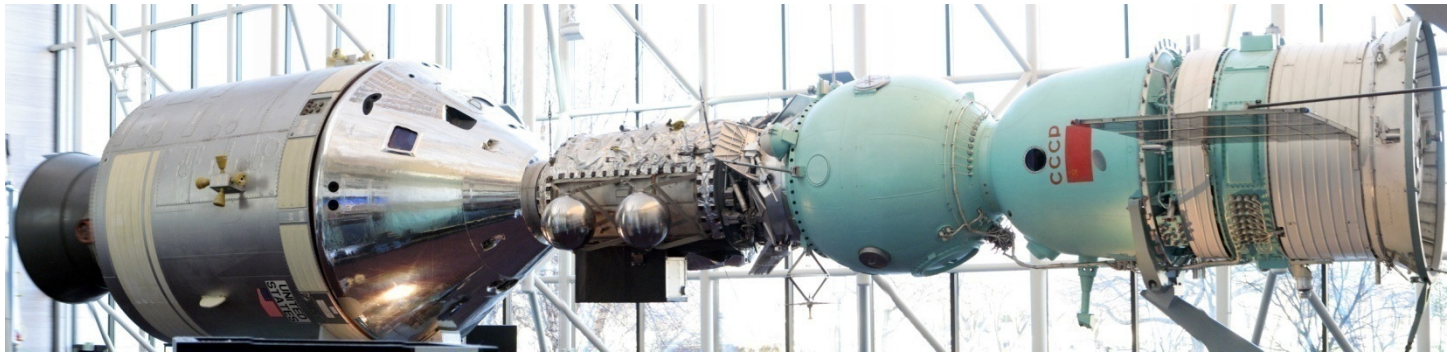
# Architecture of Distributed Systems 2011/2012

Definition of some of the used terms



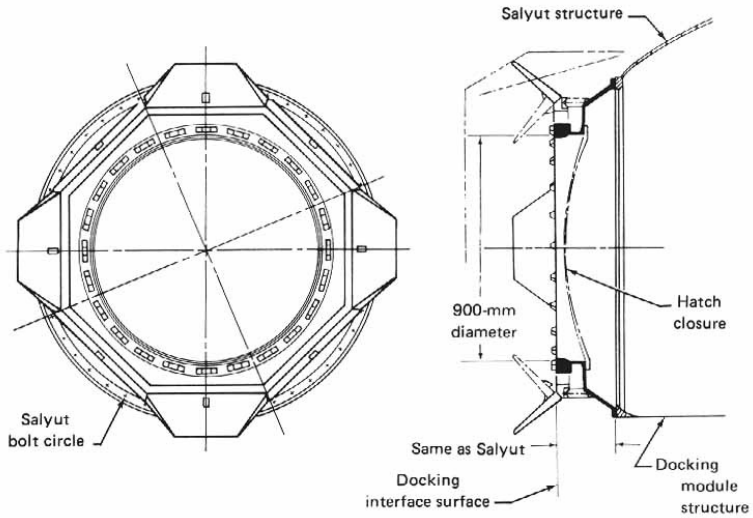
Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

# Interfaces and services



Smithsonian Air and Space museum, Washington D.C.

- Apollo-Soyuz project ('75)
  - connect Apollo and Soyuz in space
- Needs clear definition of the <u>interfaces</u>
  - structure
    - ridges, rings, edges must fit
  - functionality (service)
    - the expected and provided responses of active elements
    - flow of control
- Adapter: extra component to reconcile two interfaces
  - translation, sometimes stateholding



# Definition: service

- **Service**: *a contractually specified overall functionality (semantics) of an entity.*

- **Service quality**: extra-*functional properties of a service (e.g. speed, reliability, ...).*

- **Service interface (API)**: *actions ("primitives") and responses that make the service available; these responses can be autonomous ("events", "call-backs"). In addition, a specification that*
  - *describes their effect on state variables and parameters, as well as their results;*
  - *describes rules as how and in what sequence to call them;*
  - *describes the functional and non-functional properties of sequences of calls.*

  *(i.e., the interaction or access protocol)*

# Definition: protocol

- **Protocol:** *A formal set of rules that dictates how information exchange as well as interaction between objects (can be devices, execution threads, etc.) should take place.*

  *The rules specify*
  - *the format of the messages exchanged;*
  - *a number of different protocol states and what messages are allowed to be sent in each state; these states determine, among others, the order of the messages.*
  - *timing constraints and other quality properties, if any.*

- **Note:**
  - 'message' includes function calling
  - one may specify a protocol without being explicit about the overall service it realizes

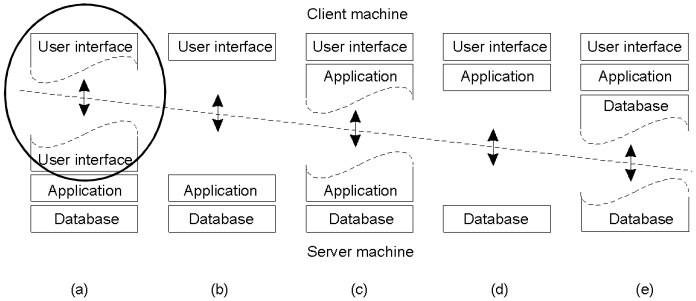
# Service example: file i/o

- *Service*: the complete functionality of being able to use files
  - store and retrieve data
  - create, destroy
- *Interface*: the calls that make this available
- *Quality*: e.g. speed, level of control (buffers), protection
- *Rules*: calling sequences (open/creat, read, ...), argument meanings, dealing with ownership, interpretation of return values
  - the access protocol

# Service e xample: The X-Window System

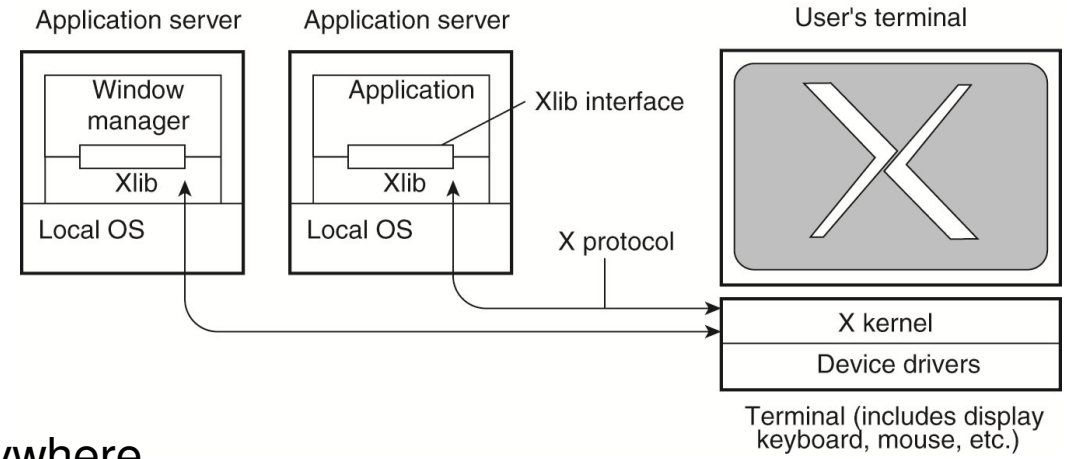
- Application servers are *clients* to the X-kernel, which is a *server* for advanced window operations

- Xlib: library that offers as a service:
  - the concepts of screen, windows, pixels, keyboard and mouse events
  - operations for manipulation of these
  - abstraction level is variable (pixels .. window)




- The *X protocol* is application independent

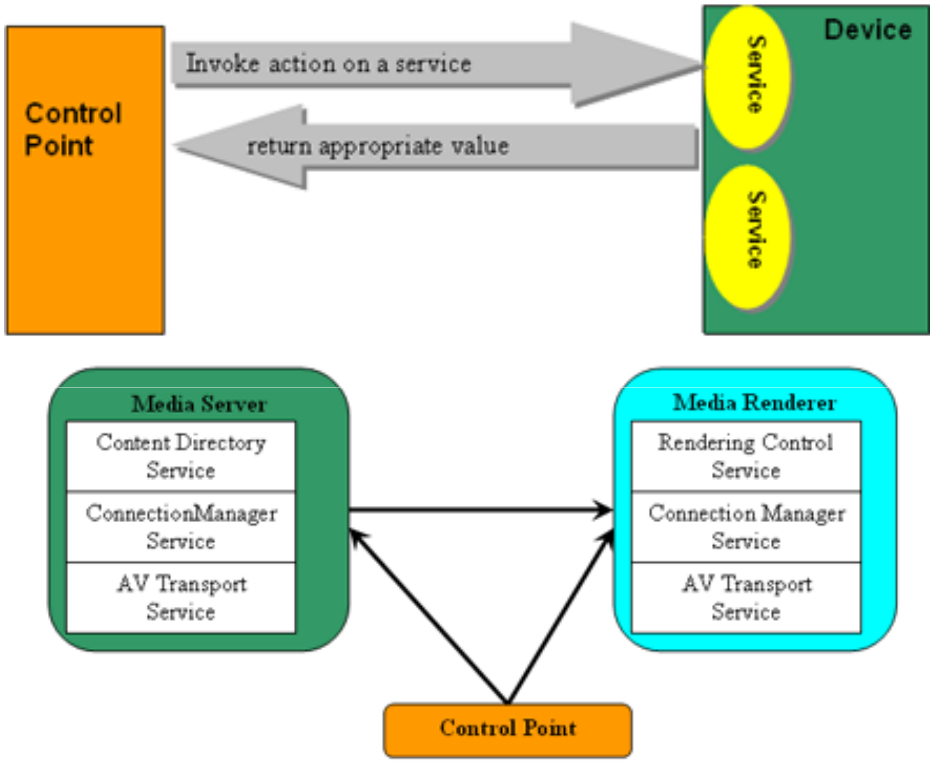
- The manager of the screen (to manage the windows) is just another client




- Application servers can run anywhere
  - enables 'thin client' organization: locally, at the X server, just user interaction

# Example: UPnP services

- UPnP services are accessed using a REST-like style
  - although there is some debate on this, see e.g. the paper by *Newmarch*
- Service implementation is entirely hidden, as are the OS and the implementation language




Interactions: control points call actions on services (top) or establish connections between services (bottom)

# Definition: carrier, binding

- A protocol can be realized directly using basic services (e.g. hardware transport, machine instructions). If not, it relies on another service and corresponding protocol, called the *carrier*.

- The functionality and properties that a protocol provides is called the *provided service*. The provided service is often specified by an API for the protocol.

- A protocol requires services from its carrier and any carrier providing these services can be used. The set of rules that specify how a protocol is mapped onto a carrier is called a *binding*.

- In general, *binding* refers to establishing a relation between a reference and a referred object.
  - this interpretation of the term is slightly different than protocol binding
  - usually, the meaning can be derived from the context

# Example: layering in communication