Architectural Patterns

Thilina Halloluwa (PhD)
tch@ucsc.cmb.ac.lk

Introduction

- We don’t invent patterns
- Found repeatedly in practice
- Has known properties that can be reused
- Describe a class of architectures

Parts of a pattern

- A context : A recurring , common situation
- A problem. The problem, appropriately generalised, that arises in the given context
- A solution. A successful architectural resolution to the problem. This can be described by
  - A set of element types (e.g. processes, objects, classes and repositories)
  - A stet of integration mechanisms (e.g. method calls, events etc)
  - A topological layout of the components
  - A set of constraints

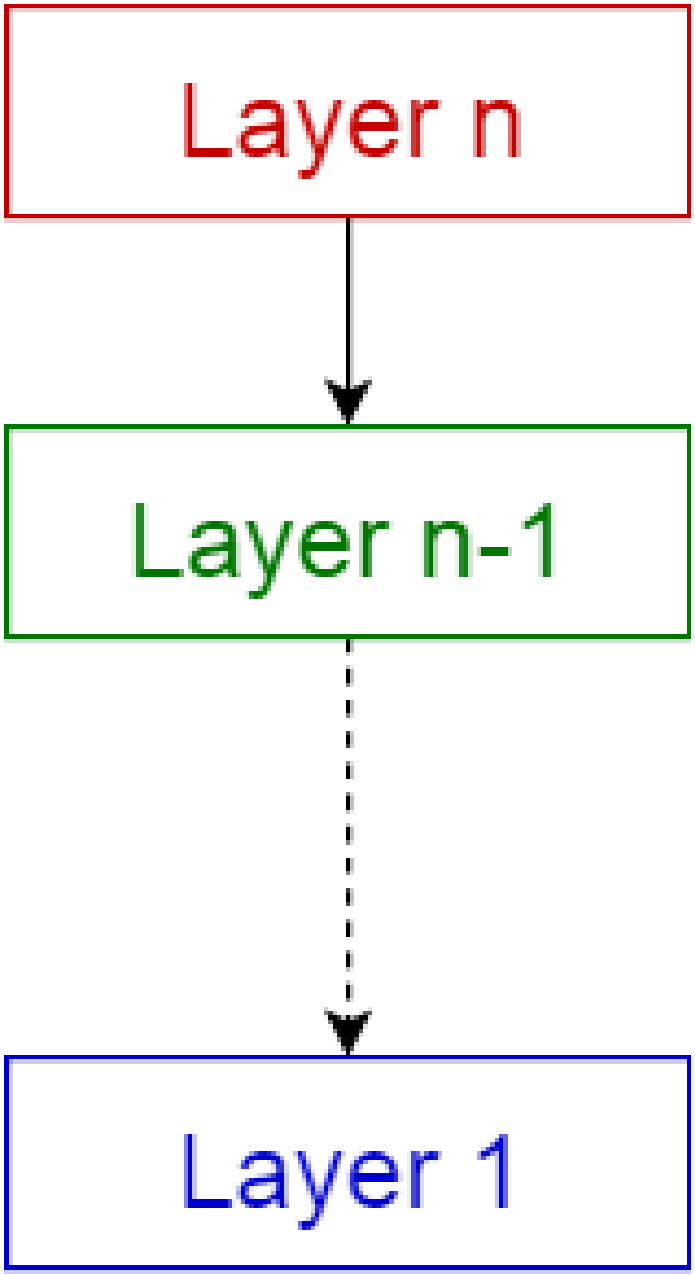
This Week

- Layered Pattern
- Broker Pattern
- Pipe-Filter Pattern
- Client-Server Pattern
- Master-slave pattern
- Service-Oriented Architecture Pattern
- Model-View-Controller Pattern

Layered Pattern

Layered pattern

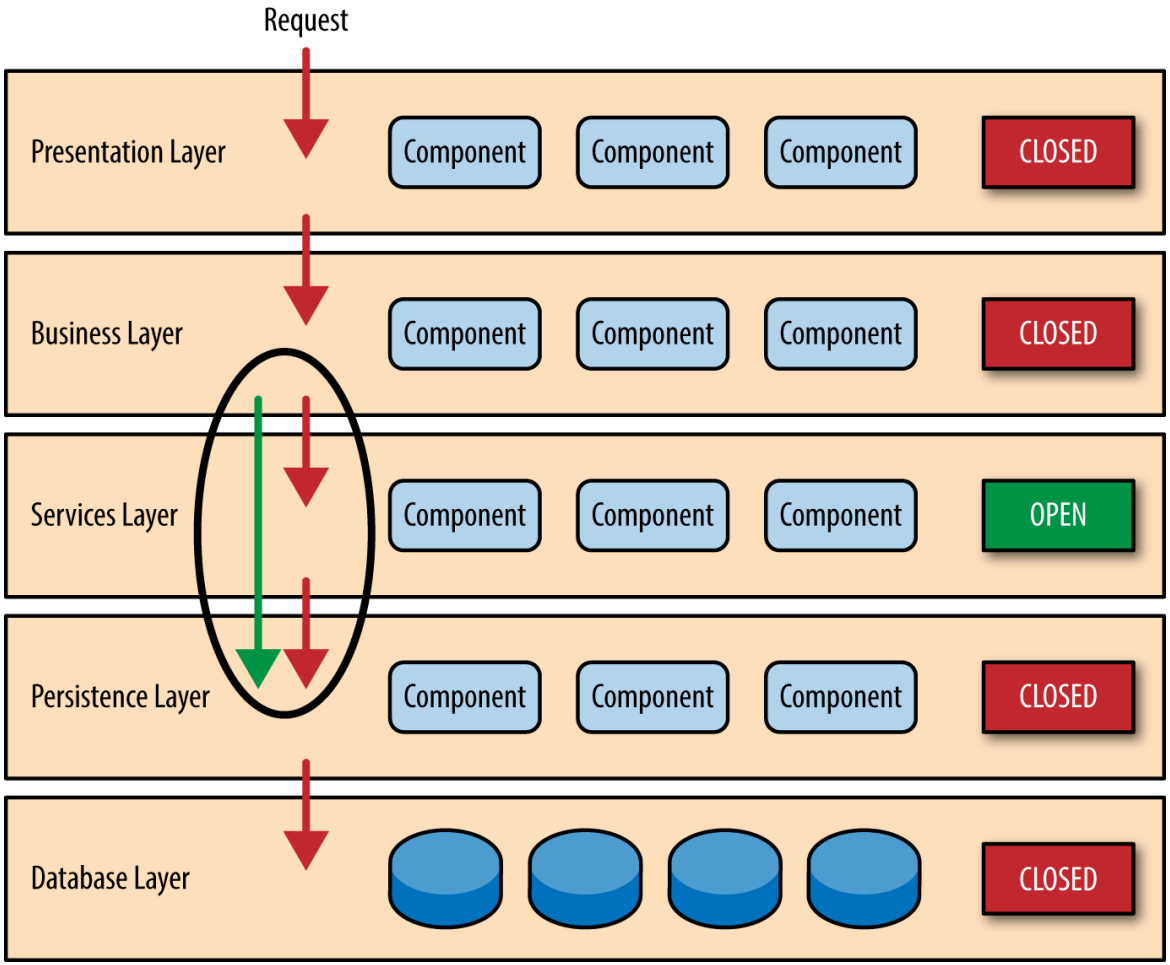
- **Context:** All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.
- **Problem:** The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.
- **Solution:** To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. This use a unidirectional communications. Layers completely partition a set of software, and each partition is exposed through a public interface.



Layered pattern

- Can be used to structure programs
- Level of abstraction
- Each layer provides services to the next higher layer.

Layered pattern



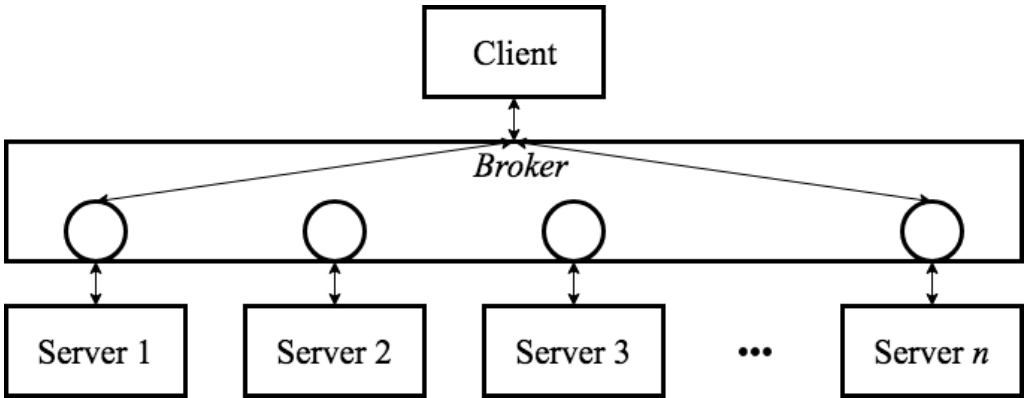
Layered Pattern

Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional allowed-to-use relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	Layer, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	Allowed to use, which is a specialization of a more generic depends-on relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	• Every piece of software is allocated to exactly one layer. • There are at least two layers (but usually there are three or more). • The allowed-to-use relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	• The addition of layers adds up-front cost and complexity to a system. • Layers contribute a performance penalty.

Broker Pattern

Broker Pattern

- **Context:** Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.
- **Problem:** How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

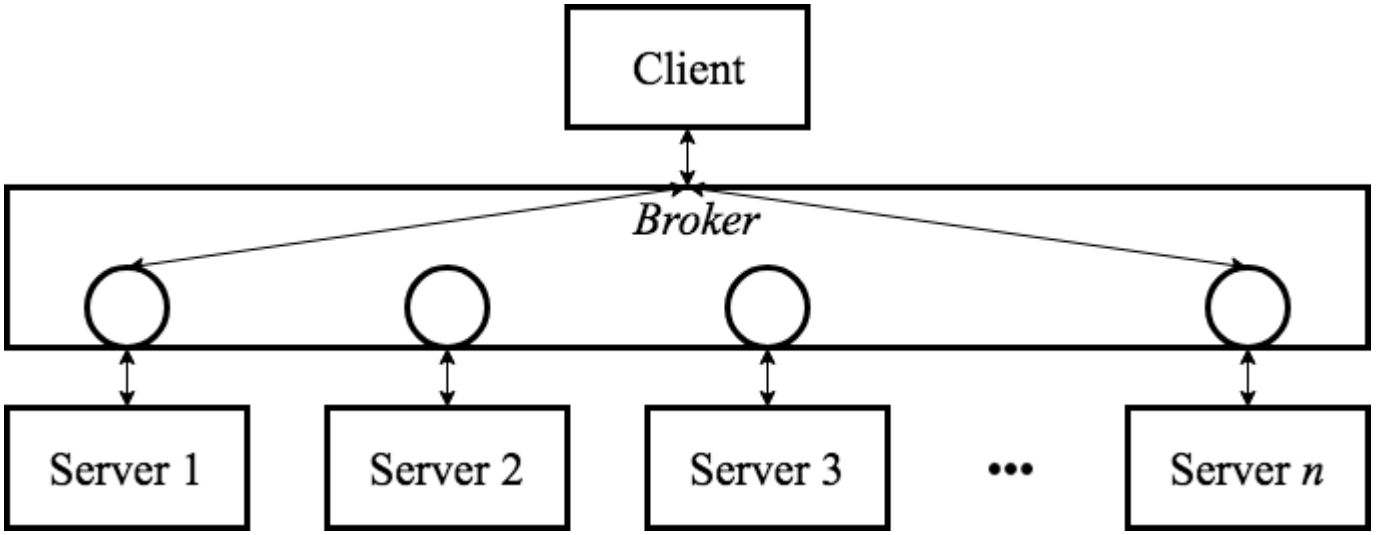


Broker Pattern

- **Context:** Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.
- **Problem:** How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?
- **Solution:** The broker pattern separates users of services (clients from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

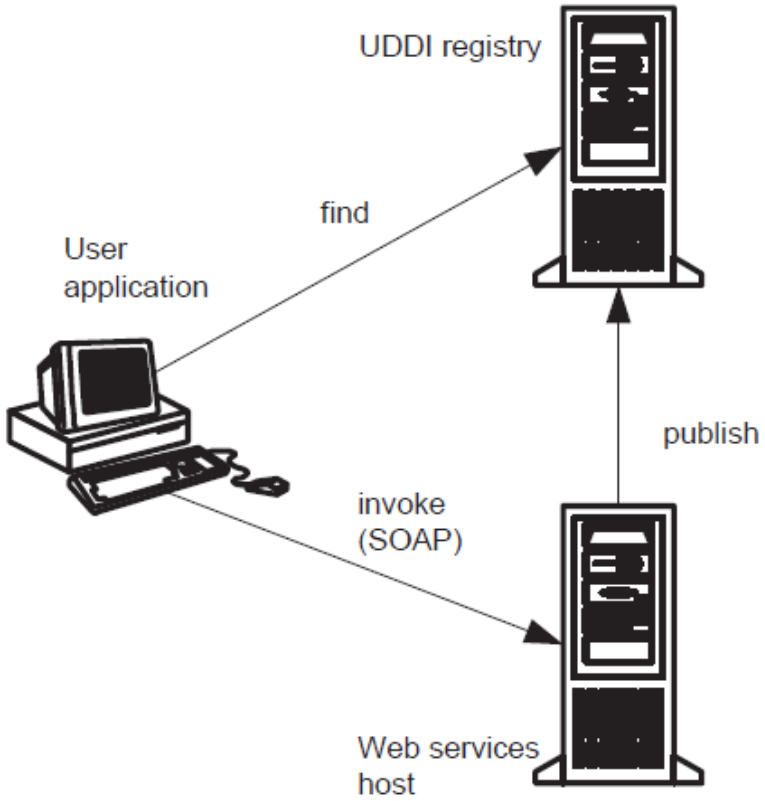
Broker Pattern

- Used to structure distributed software systems with decoupled components
- Communicates through remote procedure calls
- Components must know each other's location

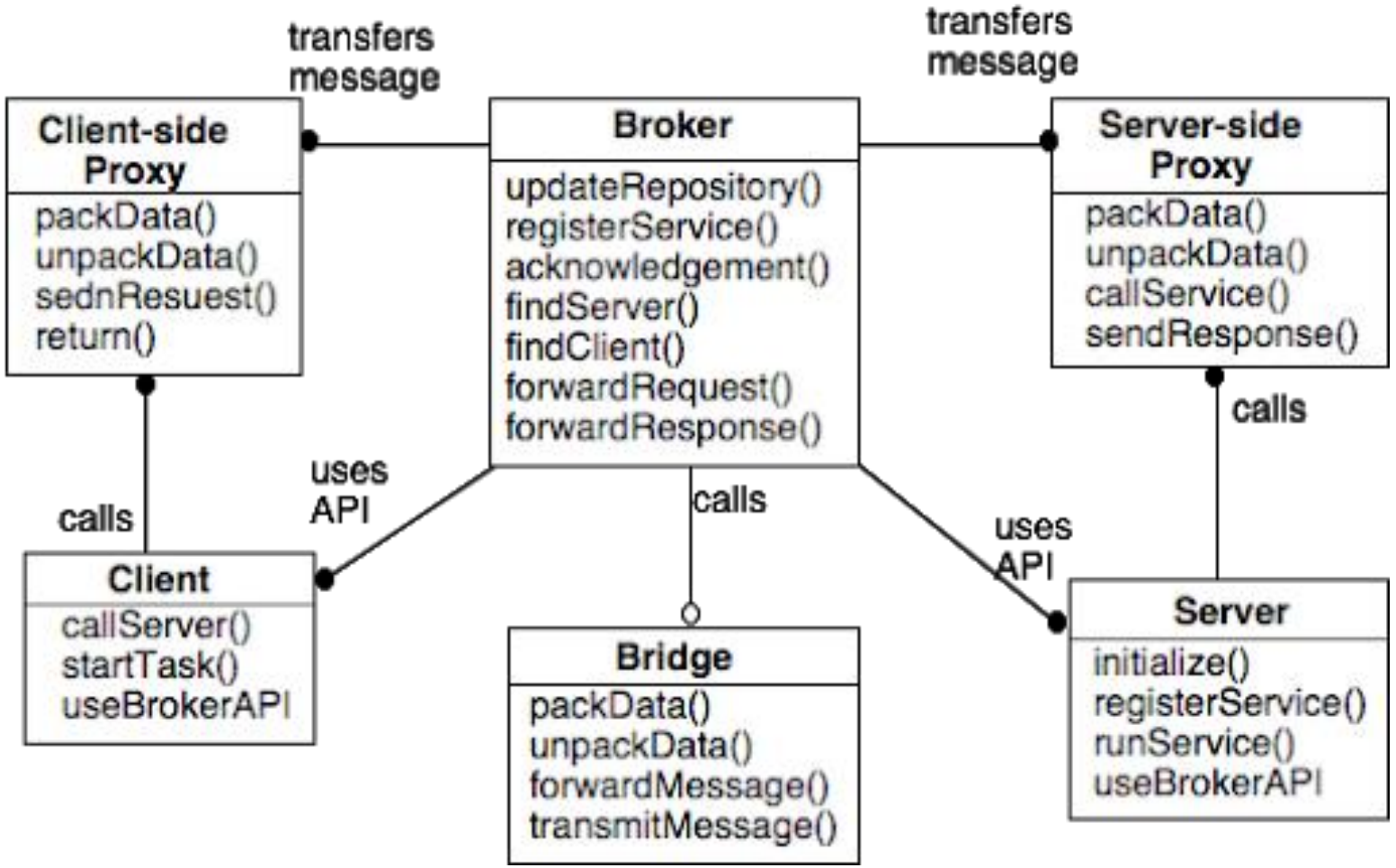


Broker Pattern

- Servers publish to a broker
- Clients request a service from the broker
- The broker then redirects the client



Broker Pattern



Broker Pattern Solution

Overview	The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers
Elements	• *Client*, a requester of services • *Server*, a provider of services • *Broker*, an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client • *Client-side proxy*, an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages • *Server-side proxy*, an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages

Broker Pattern Solution

Relations	The attachment relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
Constraints	The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server side proxy).
Weaknesses	• Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck. • The broker can be a single point of failure. • A broker adds up-front complexity. • A broker may be a target for security attacks. • A broker may be difficult to test

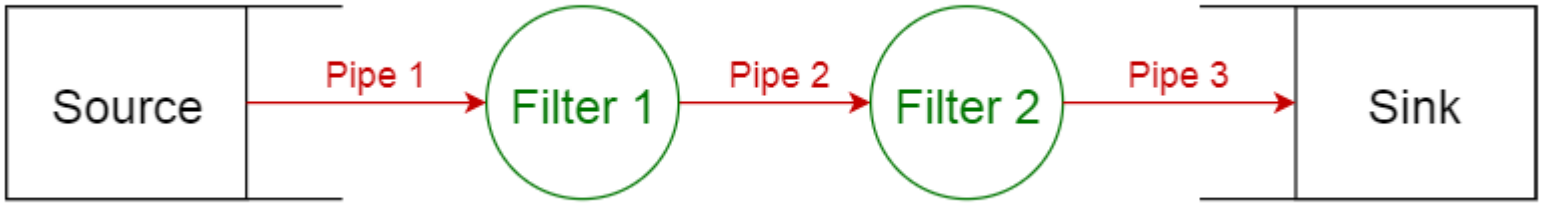
Pipe-Filter Pattern

Pipe-filter Pattern

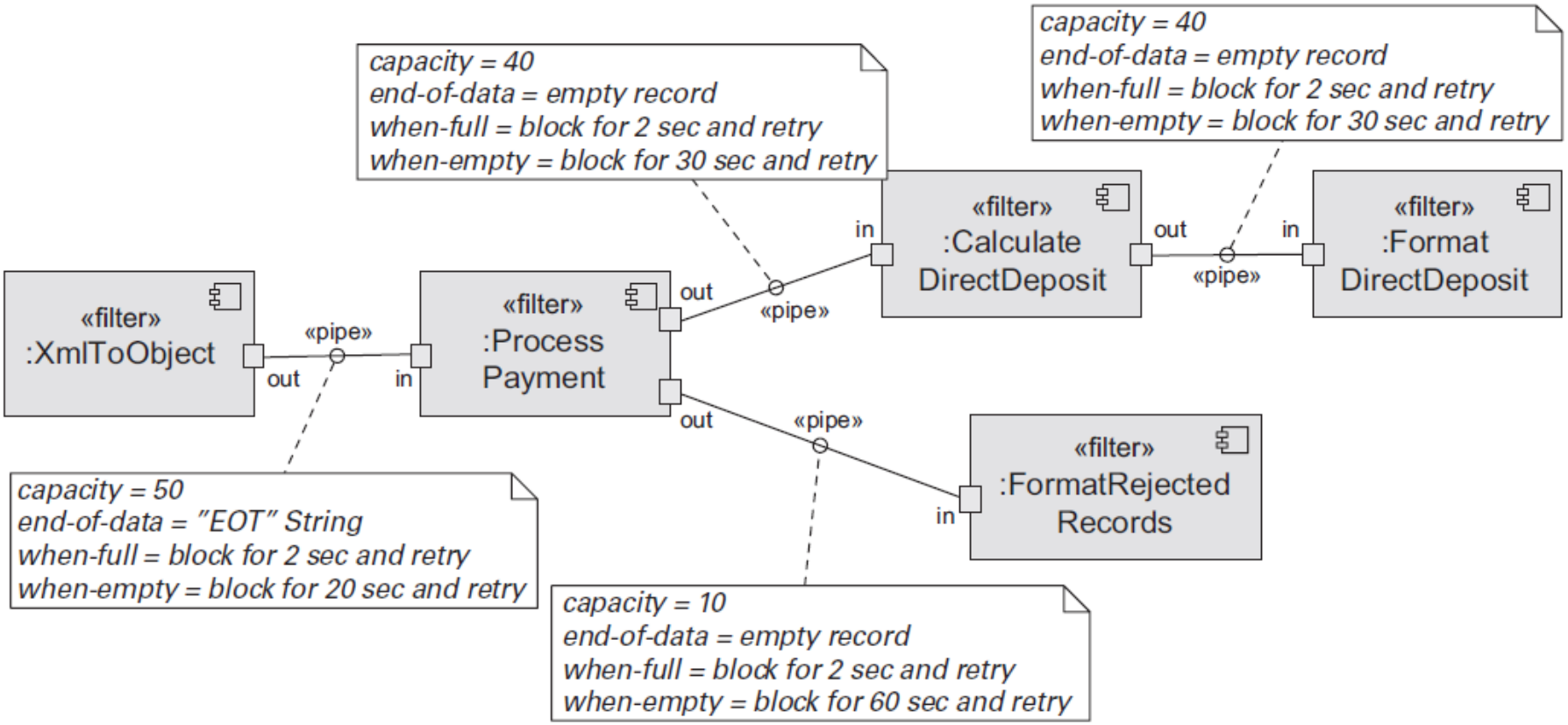
- Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these a independent, reusable parts.
- Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.
- Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Pipe-filter Pattern

- To structure systems which produce and process a stream of data
- Each processing step is enclosed within a **filter** component.
- Data to be processed is passed through **pipes**.



Pipe-filter Pattern



Pipe-filter Pattern

- Divides a task into several sequential processing steps.
- These steps are connected by the data flow
- The output of one step is the input for the next one

```
cat file | grep xyz | sort | uniq > out
```

How compilers work (Hw)

Pipe-filter Pattern Solution

Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes
Elements	**Filter**, which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. **Pipe**, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.
Relations	The attachment relation associates the output of filters with the input of pipes and vice versa.

Pipe-filter Pattern Solution

Constraints	• Pipes connect filter output ports to filter input ports. • Connected filters must agree on the type of data being passed along • Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline..
Weaknesses	• The pipe-and-filter pattern is typically not a good choice for an interactive system. • Having large numbers of independent filters can add substantial amounts of computational overhead. • Pipe-and-filter systems may not be appropriate for long-running computations.

Advantages and Disadvantages

Easy to extend.

Filters are reusable

Filters can easily be developed separately

Overhead due to data transformation

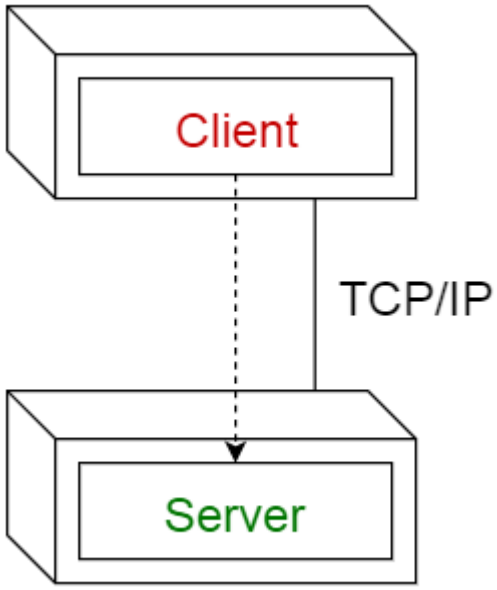
Client-Server Pattern

Client-server pattern

- **Context:** There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.
- **Problem:** By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.
- **Solution:** Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

Client-server pattern

- This pattern consists of two parties;
  - a **server** and multiple **clients**.
- Clients request services from the server and the server provides relevant services .



Client-server pattern Solution

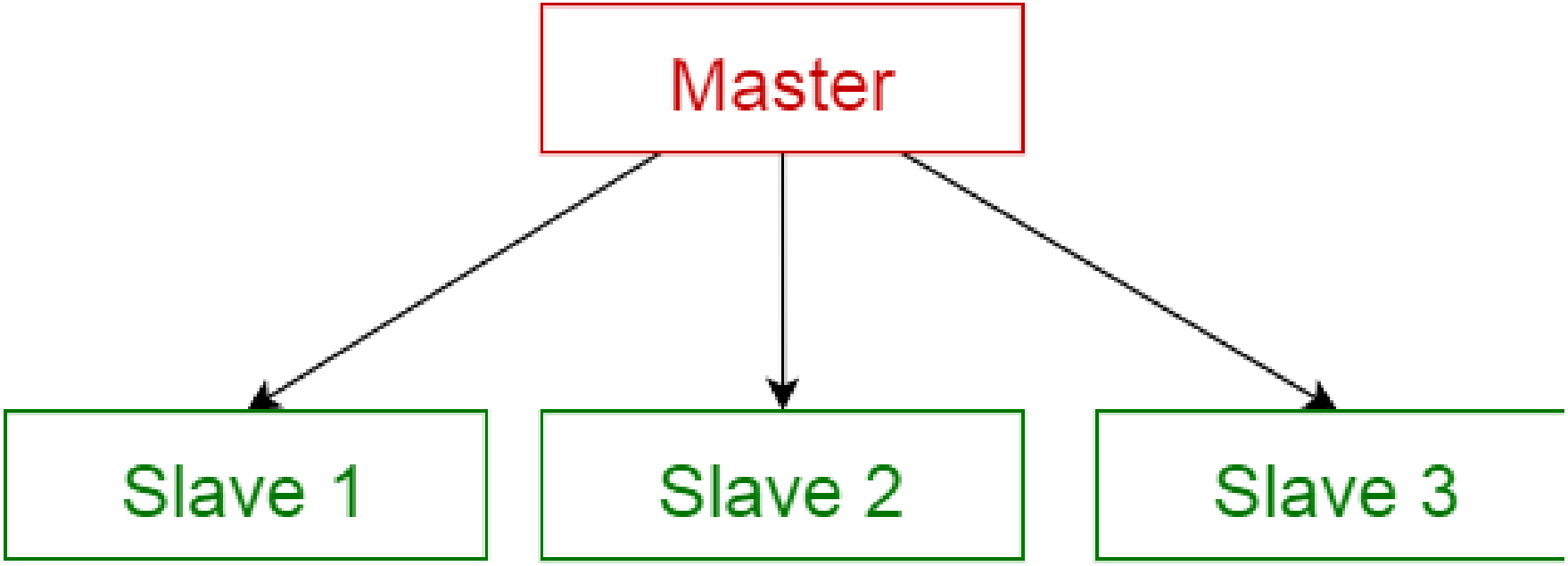
Overview	Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
Elements	*Client*, a component that invokes services of a server component. Clients have ports that describe the services they require. *Server*, a component that provides services to clients. Servers have ports that describe the services they provide. Important characteristics include information about the nature of the server ports (such as how many clients can connect) and performance characteristics (e.g., maximum rates of service invocation). *Request/reply connector*, a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.

Client-server pattern Solution

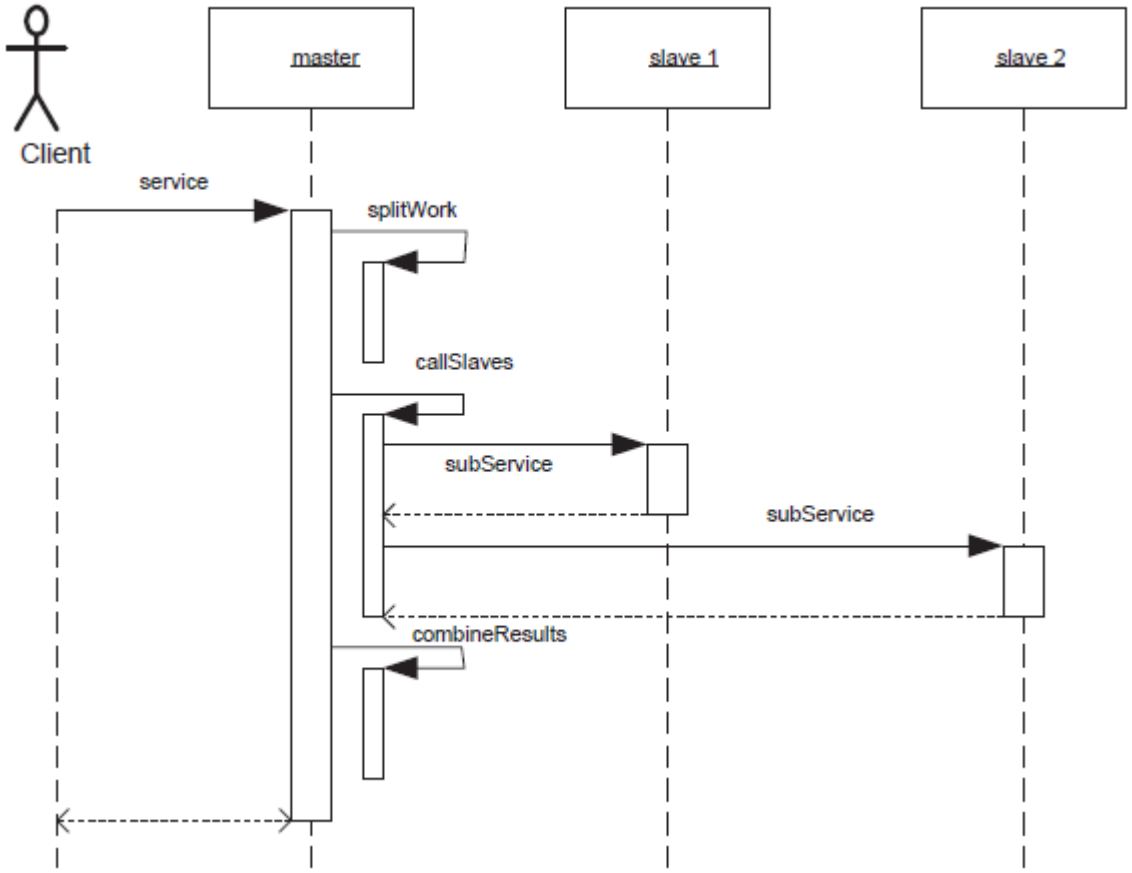
Relations	The attachment relation associates clients with servers.
Constraints	Clients are connected to servers through request/reply connectors. Server components can be clients to other servers. Specializations may impose restrictions: • Numbers of attachments to a given port • Allowed relations among servers Components may be arranged in tiers, which are logical groupings of related functionality or functionality that will share a host computing environment
Weaknesses	• Server can be a performance bottleneck. • Server can be a single point of failure. • Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Master-slave pattern

- This pattern consists of two parties
  - master
  - slaves.
- The master distributes the work
- The slave compute and send to master



Master-slave pattern



Issues (HW)

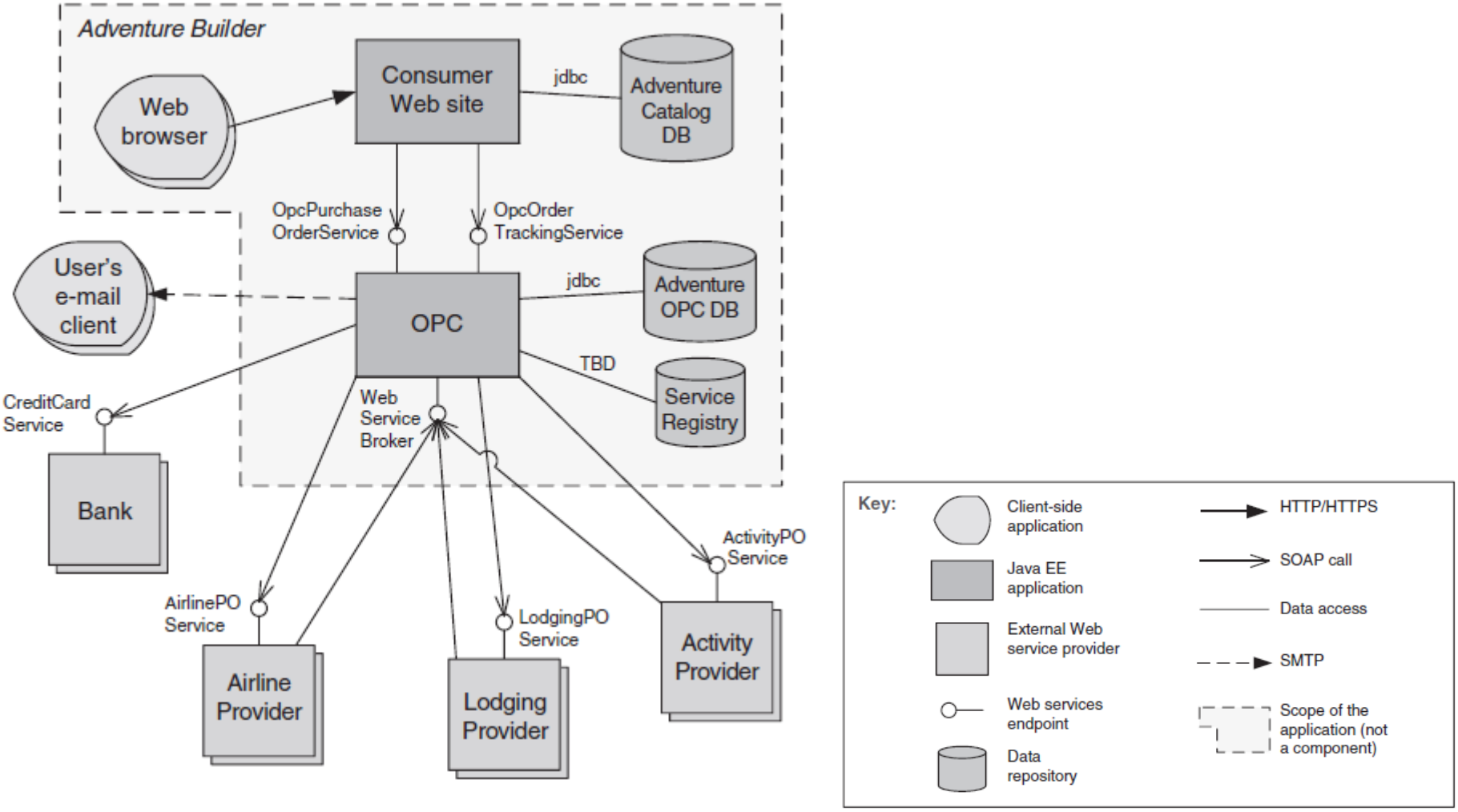
- Divide-and-conquer principle
- There is no shared state
- The latency in communication

Service-Oriented Architecture Pattern

Service-Oriented Architecture Pattern

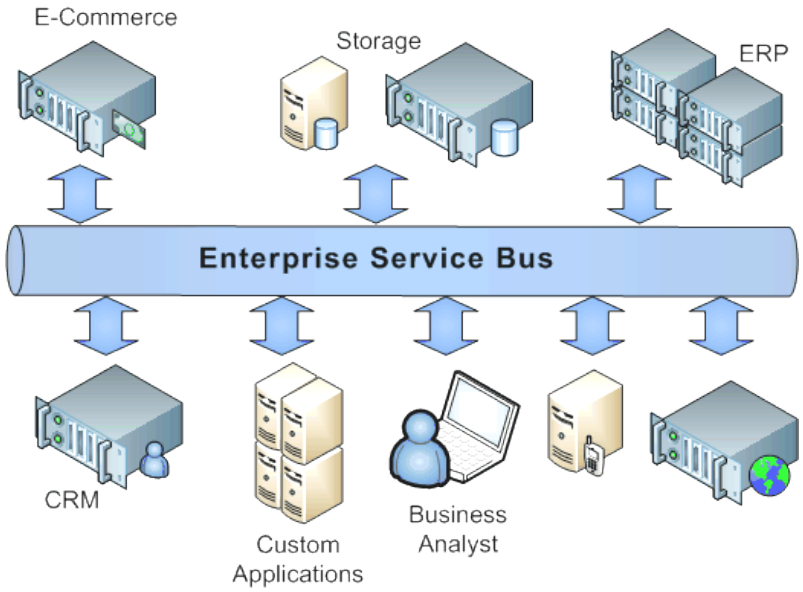
- **Context:** A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.
- **Problem:** How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?
- **Solution:** The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services. In an SOA, *service provider* components and *service consumer* components can use different implementation languages and platforms. They communicate through interfaces

Service-Oriented Architecture Pattern



SOA Elements

- Service providers and service consumers
- Enterprise service bus (ESB)
- Service registry
- Orchestration server (hw)



Service-Oriented Architecture Pattern Solution

Overview	Computation is achieved by a set of cooperating components that provide and/or consume services over a network. The computation is often described using a workflow language.
Elements	Components • Service providers, which provide one or more services through published interfaces. Concerns are often tied to the chosen implementation technology, and include performance, authorization constraints, availability, and cost. In some cases these properties are specified in a service-level agreement. • Service consumers, which invoke services directly or through an intermediary. • ESB, which is an intermediary element that can route and transform messages between service providers and consumers. • Registry of services, which may be used by providers to register their services and by consumers to discover services at runtime. • Orchestration server, which coordinates the interactions between service consumers and providers based on languages for business processes and workflows. Connectors: • SOAP connector, which uses the SOAP protocol for synchronous communication between web services, typically over HTTP. • REST connector, which relies on the basic request/reply operations of the HTTP protocol. • Asynchronous messaging connector, which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.

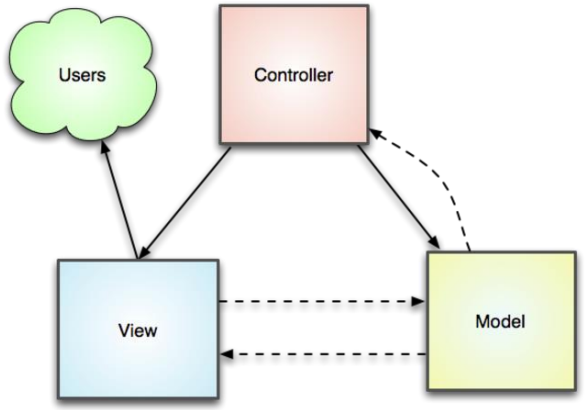
Service-Oriented Architecture Pattern Solution

Relations	Attachment of the different kinds of components available to the respective connectors
Constraints	Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
Weaknesses	• SOA-based systems are typically complex to build. • You don't control the evolution of independent services. • There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees

Model-View-Controller Pattern

Model-View-Controller Pattern

- The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components
  - The model
  - The View
  - The Controller
- Create scalable and extensible projects



MVC is not NEW

- MVC architecture first discussed in 1979 by Trygve Reenskaug
- MVC model was first introduced in 1987 in the Smalltalk programming language.
- MVC was first time accepted as a general concept, in a 1988 article

MVC Components - Model

- The model component stores data and its related logic.
- Represents data that is being transferred between controller components or any other related business logic.
- Interactions with the database (Insert, update, delete, select)

MVC Components - View

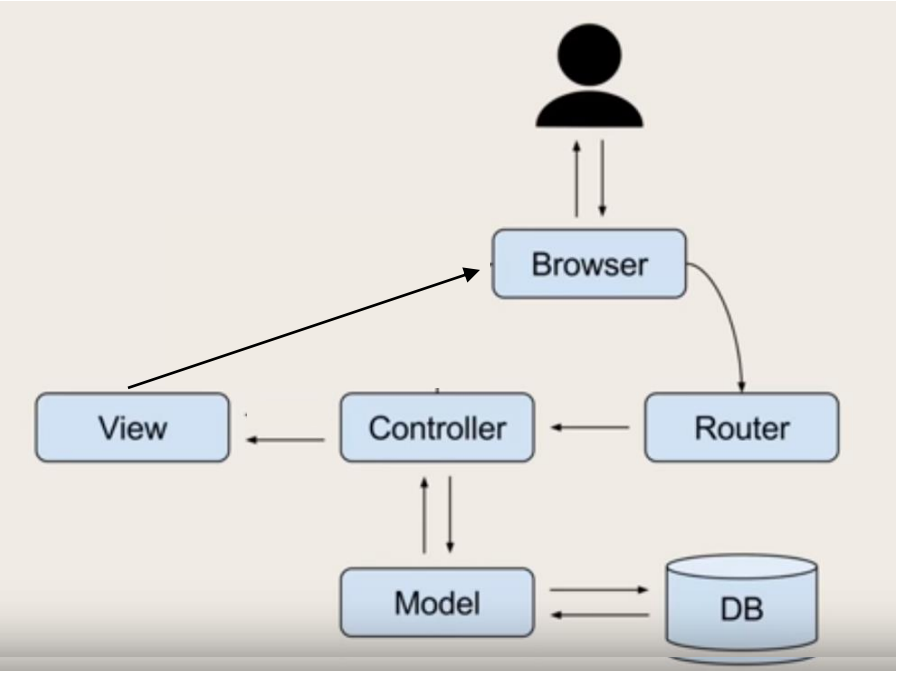
- A View is the part of the application that represents the presentation of data.
- Views are created by the data collected from the model data

MVC Components - Controller

- The Controller is that part of the application that handles the user interaction
- Controller interprets the inputs from the user, informing model and the view to change as appropriate.
  - A Controller send's commands to the model to update its state
  - The controller also sends commands to its associated view to change the view's presentation

MVC Example - Restaurant

- You visit a restaurant and wait for the waiter to come on.
- The waiter takes the order and pass it to the chef at the kitchen along with your table number.
- The chef then prepares your meal by taking ingredients out of the refrigerator
- Once done, the chef handover the food to the waiter.
- Waiter looks at the table number of the meal and serve you




```
http://MyAPP.com/users/profile/1

/routes
    users/profile/:id = Users.getProfile(id)

/Controllers

    Class Users {
        function getProfile(id) {
            profile=this.UserModel.getProfile(id);
            renderView('users/profile', profile);
        }
    }

/models

    Class UserModel{
        function getProfile(id){
            data=this.db.get('select * from users where id=id')
            return data;
        }
    }

/Views

    /users
        /getProfile

        <h1> {{profile.name}} </h1>
        <ul>
            <li> Email: {{profile.email}} </li>
            <li> Phone: {{profile.phone}} </li>
        </ul>
```

Model-View-Controller Pattern

- **Context:** User interface software is typically the most frequently modified portion of an interactive application. For this reason it is important to keep modifications to the user interface software separate from the rest of the system. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.
- **Problem:** How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?
- **Solution:** The model-view-controller (MVC) pattern separates application functionality into three kinds of components:, model, view and controller

Model-View-Controller Pattern Solution

Overview	The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
Elements	Model, View, and Controller,
Relations	The notifies relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	• There must be at least one instance each of model, view, and controller..
Weaknesses	• The complexity may not be worth it for simple user interfaces. • The model, view, and controller abstractions may not be good fits for some user interface toolkits

A *controller*, which mediates between the *model* and the *view* and manages the notifications of state changes !!!!

Homework ☺

- Peer-to-Peer Pattern (page 220)
- Publish-Subscribe Pattern (page 226)
- Map-Reduce Pattern (page 232)

Thank You

tch@ucsc.cmb.ac.lk