Lecture Week 6

Semester 2 2023

UNIVERSITY OF CANBERRA

SOFTWARE SYSTEMS ARCHITECTURE UG/G (11491/8746)

Richa Awasthy
richa.awasthy@canberra.edu.au

DISTINCTIVE BY DESIGN

ACKNOWLEDGEMENT OF COUNTRY

The University of Canberra acknowledges the Ngunnawal people as the traditional custodians of the land upon which the university's main campus sits, and pays respect to all Elders past and present.

WEEK 6: AGENDA

UNIVERSITY OF CANBERRA

1. Architectural Tactics & Patterns
    - Main Patterns
    - Patterns & Tactics

1. Architectural Tactics & Patterns

Learning Outcomes

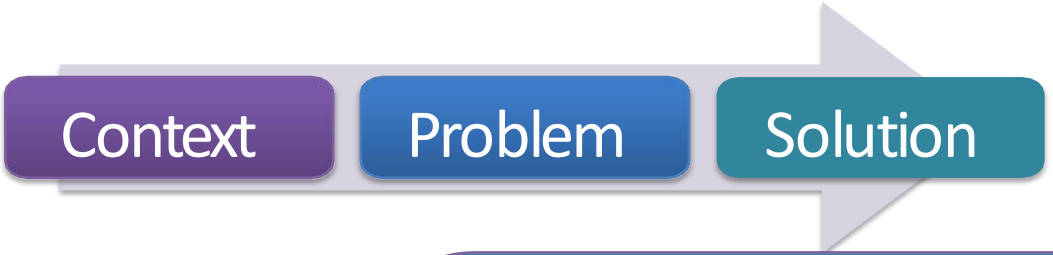
UNIVERSITY OF CANBERRA

- Quality Attributes

1. Demonstrate a firm understanding of the principles of software architecture, architectural best-practices, and how architecture is used in modern software engineering;
2. Understand the role of a software architect in software engineering practice;
3. Examine and compare various architecture styles and solutions;
4. Design an architecture that reflects and balances the different needs of its stakeholders; and
5. Communicate the architecture to stakeholders and demonstrate that it has met their requirements.

Architectural Patterns

An architectural pattern establishes a relationship between:



Context:

- A recurring, common situation in the world that gives rise to a problem

Problem:

- The problem, appropriately generalized, that arises in the given context

Solution:

- Successful architectural resolution to the problem
- The solution for a pattern is determined by:
  - A set of element types (e.g., data repositories, processes, and objects)
  - A set of interaction mechanisms or connectors
  - (e.g., method calls, message bus)
  - A topological layout of the components
  - A set of semantic constraints: topology, behaviour, and interaction mechanisms

Main Patterns

Layer Pattern

Context:

- All complex systems experience the need to develop and evolve portions of the system independently
- 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:

- Software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts
- Support portability, modifiability, and reuse

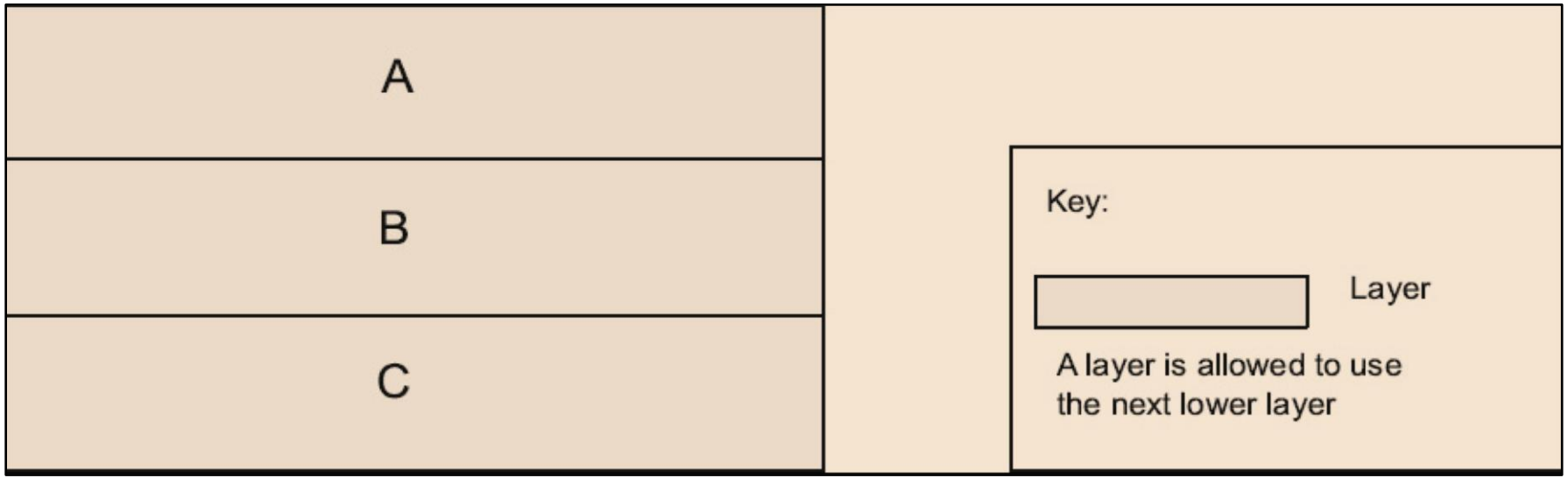
Solution:

- 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
- Layers completely partition a set of software; each partition is exposed through a public interface

Layer Pattern Solution

- 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
- Elements: *Layer*, a kind of module; the description of a layer should define what modules the layer contains
- Relations: *Allowed to use*; the design should define what the layer usage rules are 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

Layer Pattern Example



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?
- How can we make 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 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 marshalling, sending, and unmarshalling of messages
  - Server-side proxy, an intermediary that manages the actual communication with the broker, including marshalling, sending, and unmarshaling of messages

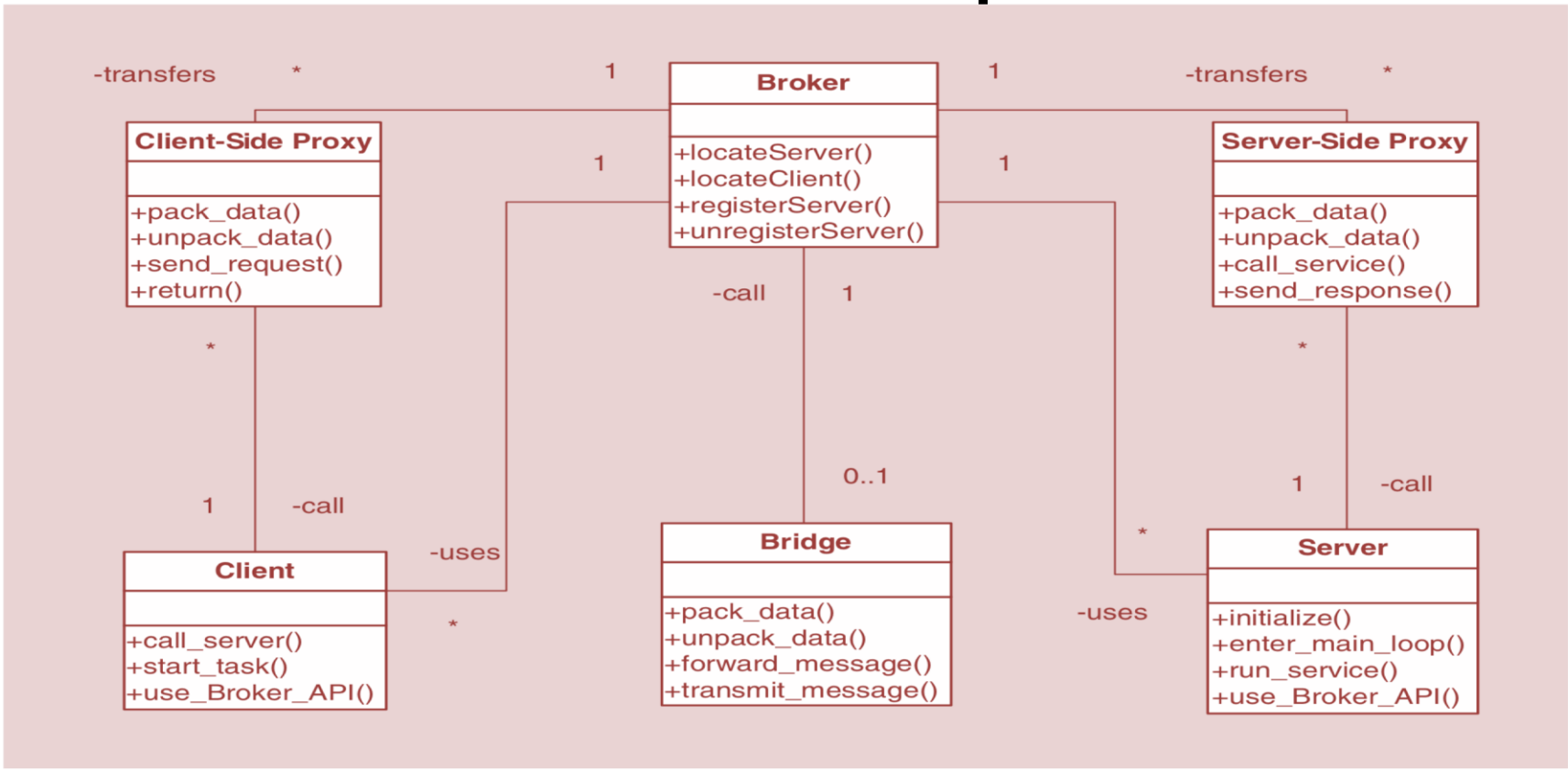
Broker Solution

- 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

Broker Example



Model-View-Controller Pattern

Context:

- User interface software is typically the most frequently modified portion of an interactive application
- 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 still be responsive for user input, or for changes in the underlying application's data?
- How can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

Solution:

- MVC separates application functionality into 3 kinds of components:
  - Model: contains the application's data
  - View: displays some portion of the underlying data & interacts with the user
  - Controller: mediates between the model and the view

MVC Solution

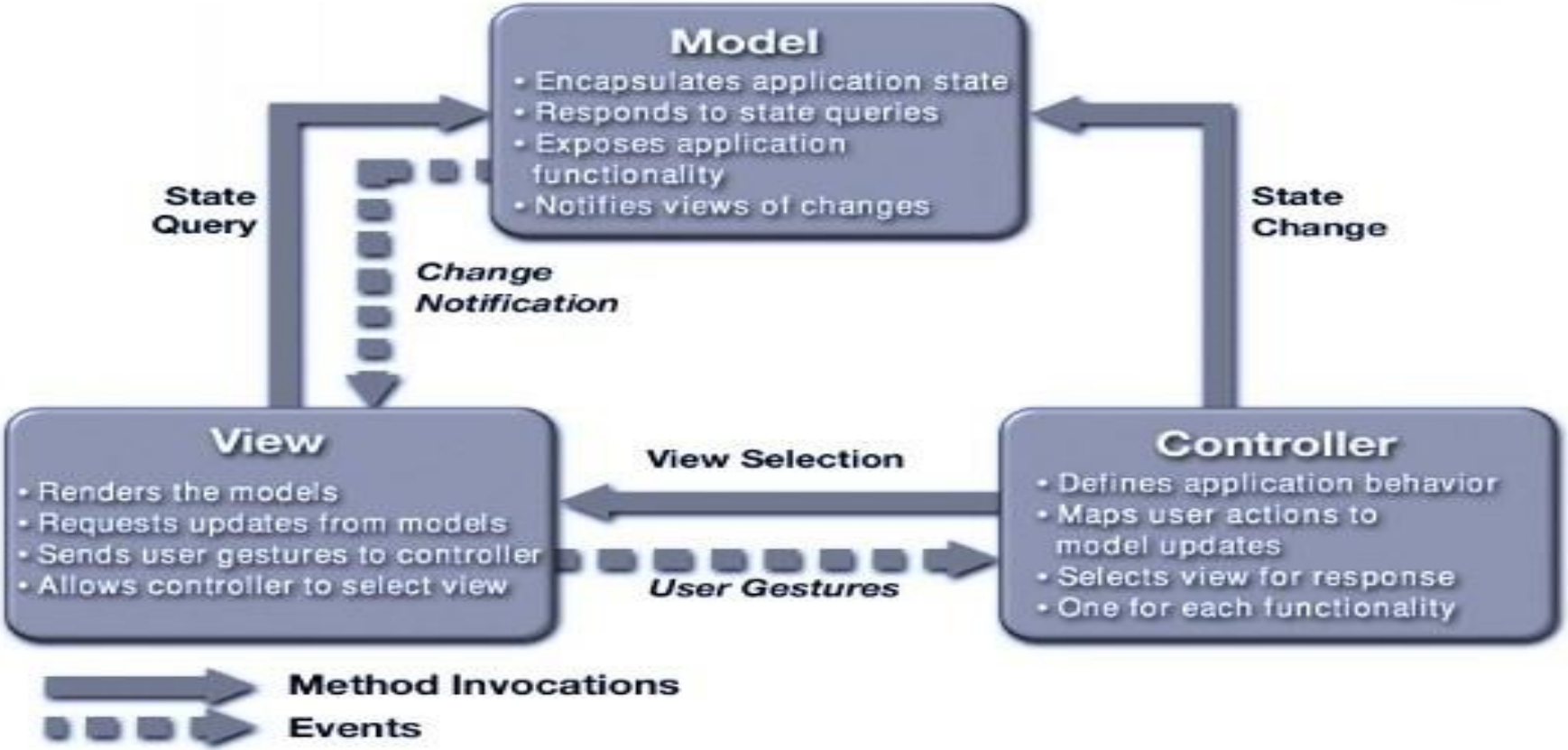
- Overview: The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view

- Elements:
  - The model is a representation of the application data or state, and it contains (or provides an interface to) application logic
  - The view is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both
  - The controller manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view

MVC Solution

- 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
  - The model component should not interact directly with the 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

MVC Example



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
  - This is done by factoring out common services and having to modify these in a single location, or a small number of locations
- Improve scalability & availability by centralizing the control of resources and services, while distributing resources 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

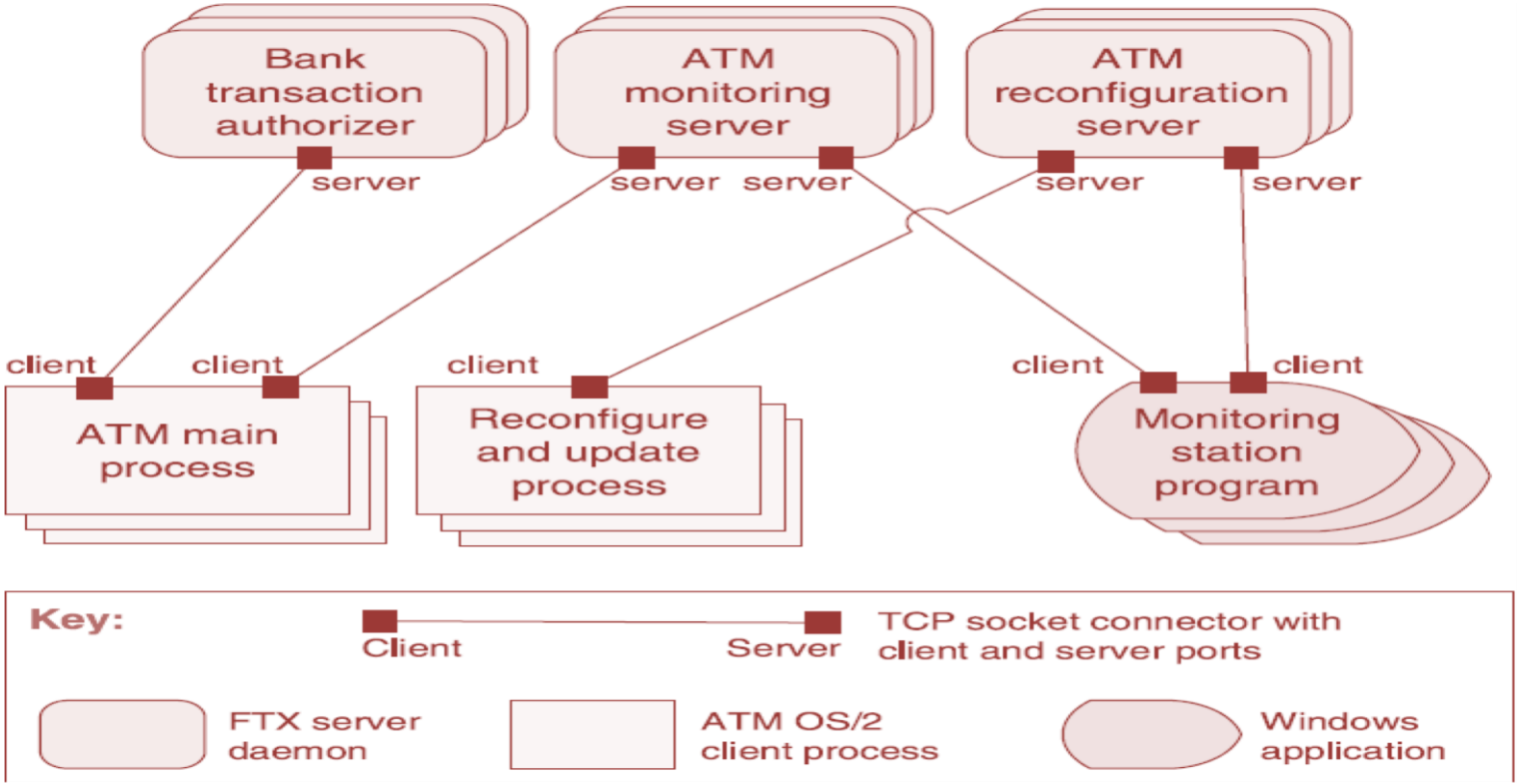
Client-Server Solution

- 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
  - 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 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.
- **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.

Client-Server Example



Peer-to-Peer Pattern

Context:

- Distributed computational entities need to cooperate and collaborate to provide a service to a distributed community of users
- Each entity is considered equally important in terms of initiating an interaction and each of which provides its own resources

Problem:

- How can a set of "equal" distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?

Solution:

- In the P2P pattern, components directly interact as peers
- All peers are "equal" and no peer or group of peers can be critical for the health of the system
- Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern

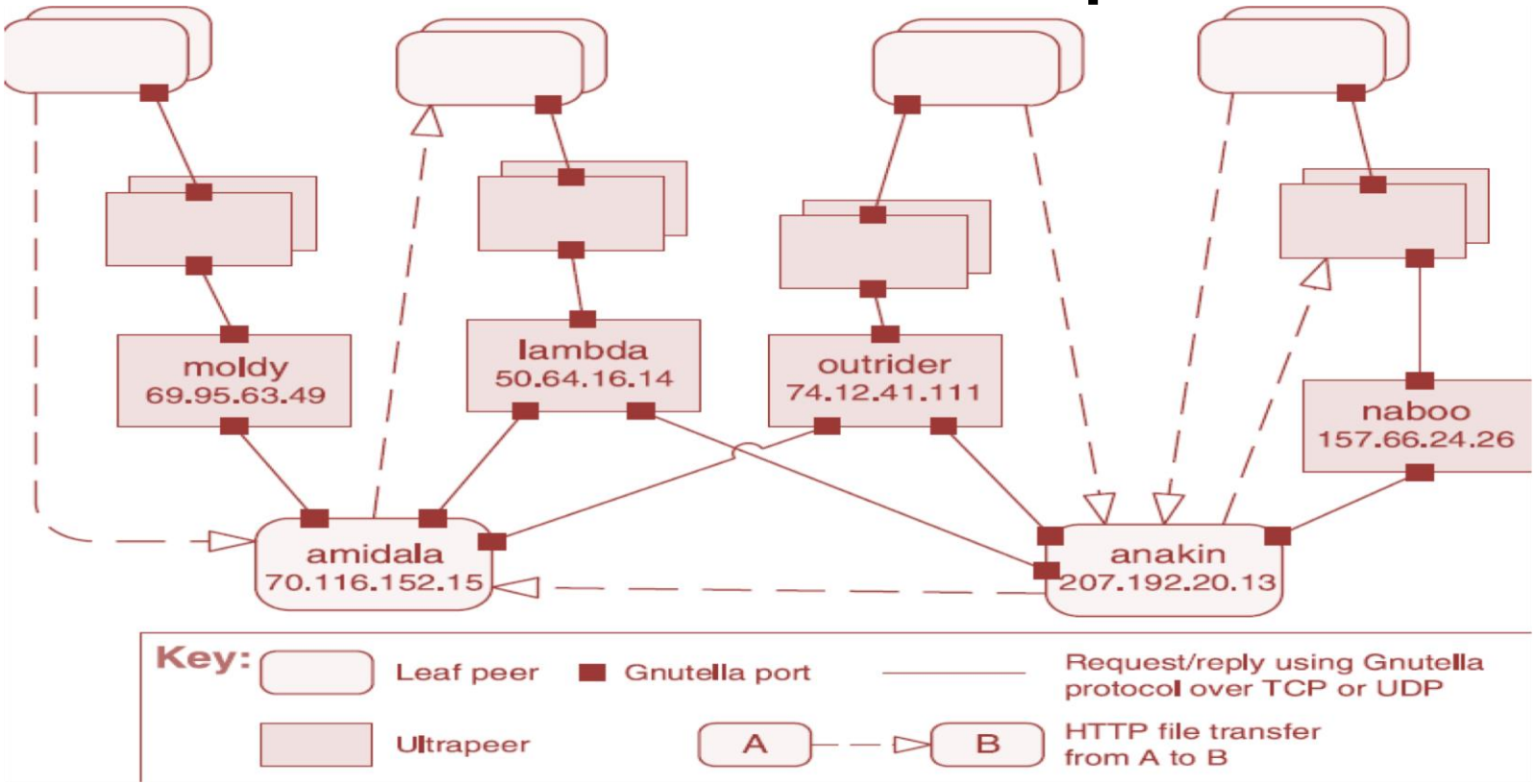
Peer-to-Peer Solution

- **Overview:** Computation is achieved by cooperating peers that request service from and provide services to one another across a network
- **Elements:**
  - Peer, which is an independent component running on a network node; special peer components can provide routing, indexing, and peer search capability
  - Request/reply connector, which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with
- **Relations:** The relation associates peers with their connectors; attachments may change at runtime

Peer-to-Peer Solution

- **Constraints**: Restrictions may be placed on the following:
  - The number of allowable attachments to any given peer
  - The number of hops used for searching for a peer
  - Which peers know about which other peers
  - Some P2P networks are organized with star topologies, in which peers only connect to supernodes
- **Weaknesses:**
  - Managing security, data consistency, data/service availability, backup, and recovery are all more complex
  - Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability

Peer-to-Peer Example



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

Service Oriented Architecture Solution

- Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network

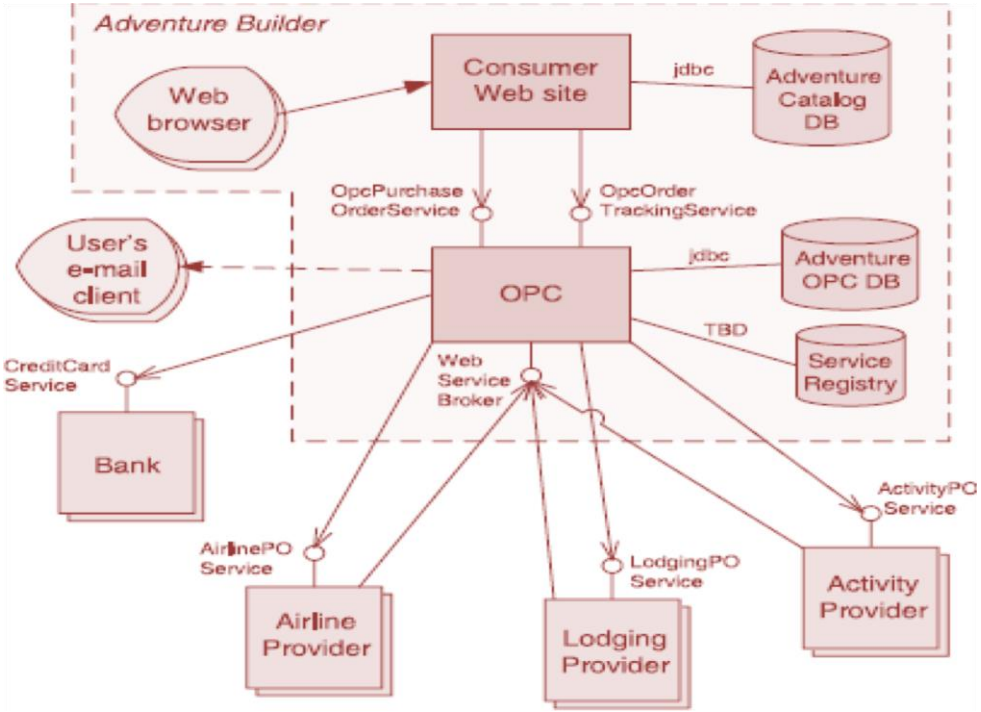
Service Oriented Architecture Solution

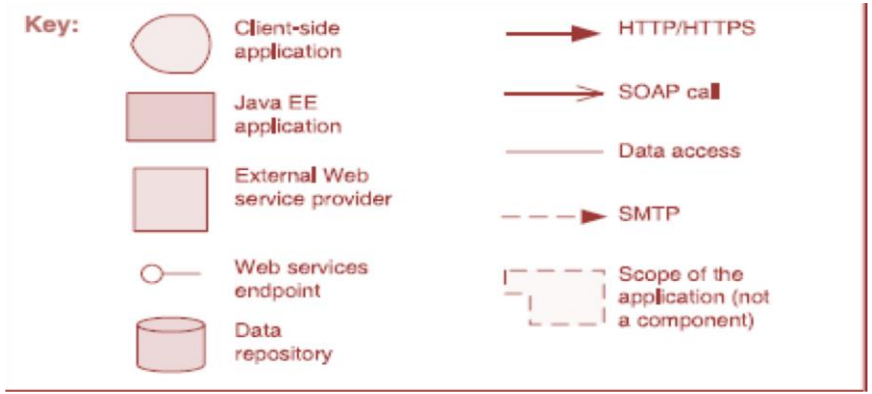
- **Elements:**
  - Components:
    - Service providers
    - Service consumers
- **ESB**, which is an intermediary element that can route and transform messages between service providers and consumers
- **Registry of services**, which may be used by providers to register their services and by consumers 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
  - REST connector
  - Asynchronous messaging connector

Service Oriented Architecture 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

Service Oriented Architecture Example





Shared-Data Pattern

Context:

- Various computational components need to share and manipulate large amounts of data
- This data does not belong solely to any one of those components

Problem:

- How can systems store and manipulate persistent data that is accessed by multiple independent components?

Solution:

- In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple data accessors and at least one shared-data store
- Exchange may be initiated by the accessors or the data store
- The connector type is data reading and writing

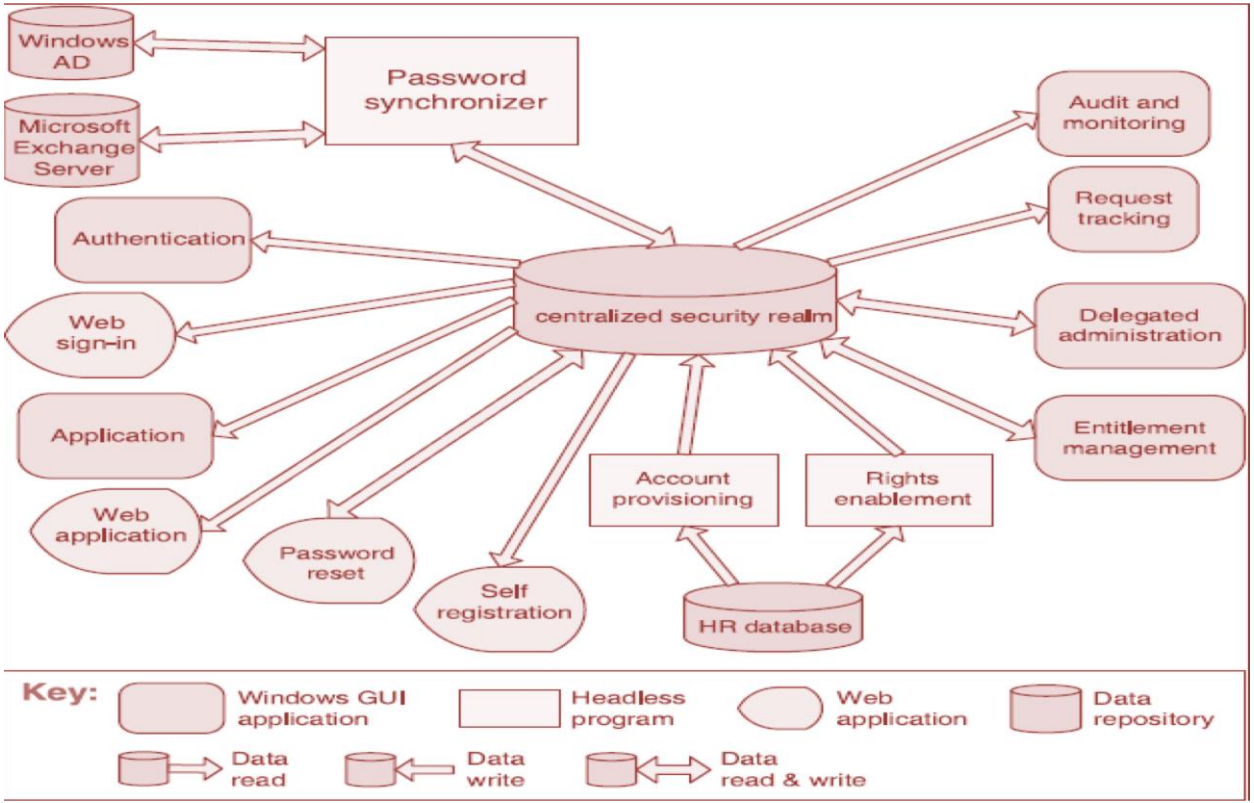
Shared Data Solution

- **Overview:** Communication between data accessors is mediated by a shared data store; control may be initiated by the data accessors or the data store; Data is made persistent by the data store
- **Elements:**
  - Shared-data store, concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted
  - Data accessor component
  - Data reading and writing connector

Shared Data Solution

- Relations: Attachment relation determines which data accessors are connected to which data stores
- Constraints: Data accessors interact only with the data store(s)
- Weaknesses:
  - The shared-data store may be a performancebottleneck
  - The shared-data store may be a single point of failure
  - Producers and consumers of data may be tightly coupled

Shared Data Example



Map-Reduce Pattern

Context:

- Businesses have a pressing need to quickly analyse enormous volumes of data they generate or access, at petabyte scale

Problem:

- For many applications with ultra-large data sets, sorting the data and then analysing the grouped data is sufficient
- The map-reduce pattern solves efficiently the problem of performing a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done

Solution:

- The map-reduce pattern requires three parts:
  - Specialized infrastructure: allocate software to hardware nodes in a massively parallel computing environment and handles sorting the data as needed
  - Map: filters the data to retrieve those items to be combined
  - Reduce: combines the results of the map

Map-Reduce Solution

Overview: The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors

Elements:

- Map is a function with multiple instances deployed across multiple processors that extract and transformation portions of the analysis
- Reduce is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load
- The infrastructure is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure

Map-Reduce Solution

Relations:

- Deploy on is the relation between an instance of a map or reduce function and the processor onto which it is installed
- Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce

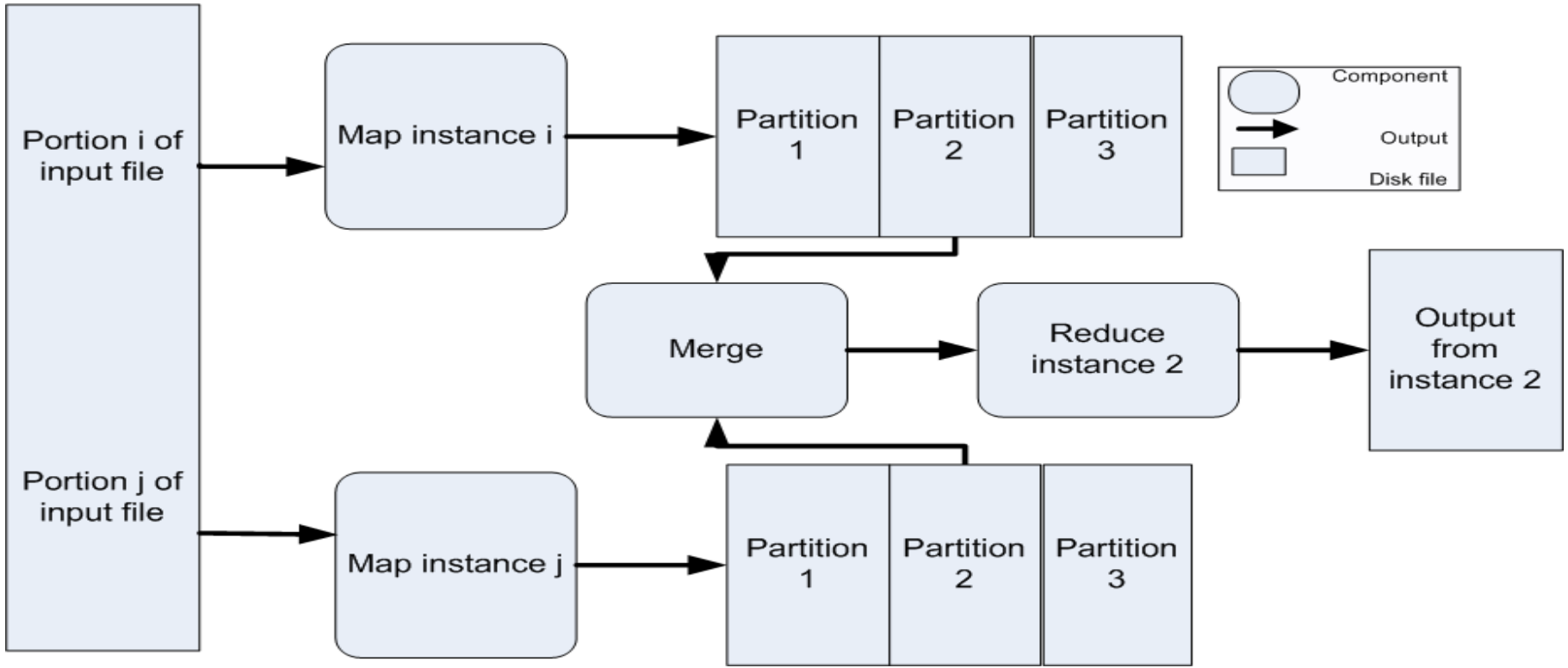
Constraints:

- The data to be analyzed must exist as a set of files
- Map functions are stateless and do not communicate with each other
- The only communication between map reduce instances is the data emitted from the map instances as <key, value> pairs

Weaknesses:

- If you do not have large data sets, the overhead of map-reduce is not justified
- If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost
- Operations that require multiple reduces are complex to orchestrate

Map-Reduce Example



Multi-Tier Pattern

Context:

- In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets

Problem:

- How can we split the system into a number of computationally independent execution structures (groups of software and hardware) connected by some communications media

Solution:

- The execution structures of many systems are organized as a set of logical groupings of components
- Each grouping is termed a tier

Multi-Tier Solution

Overview: The execution structures of many systems are organized as a set of logical groupings of components; each grouping is termed a tier

Elements:

- Tier, which is a logical grouping of software components

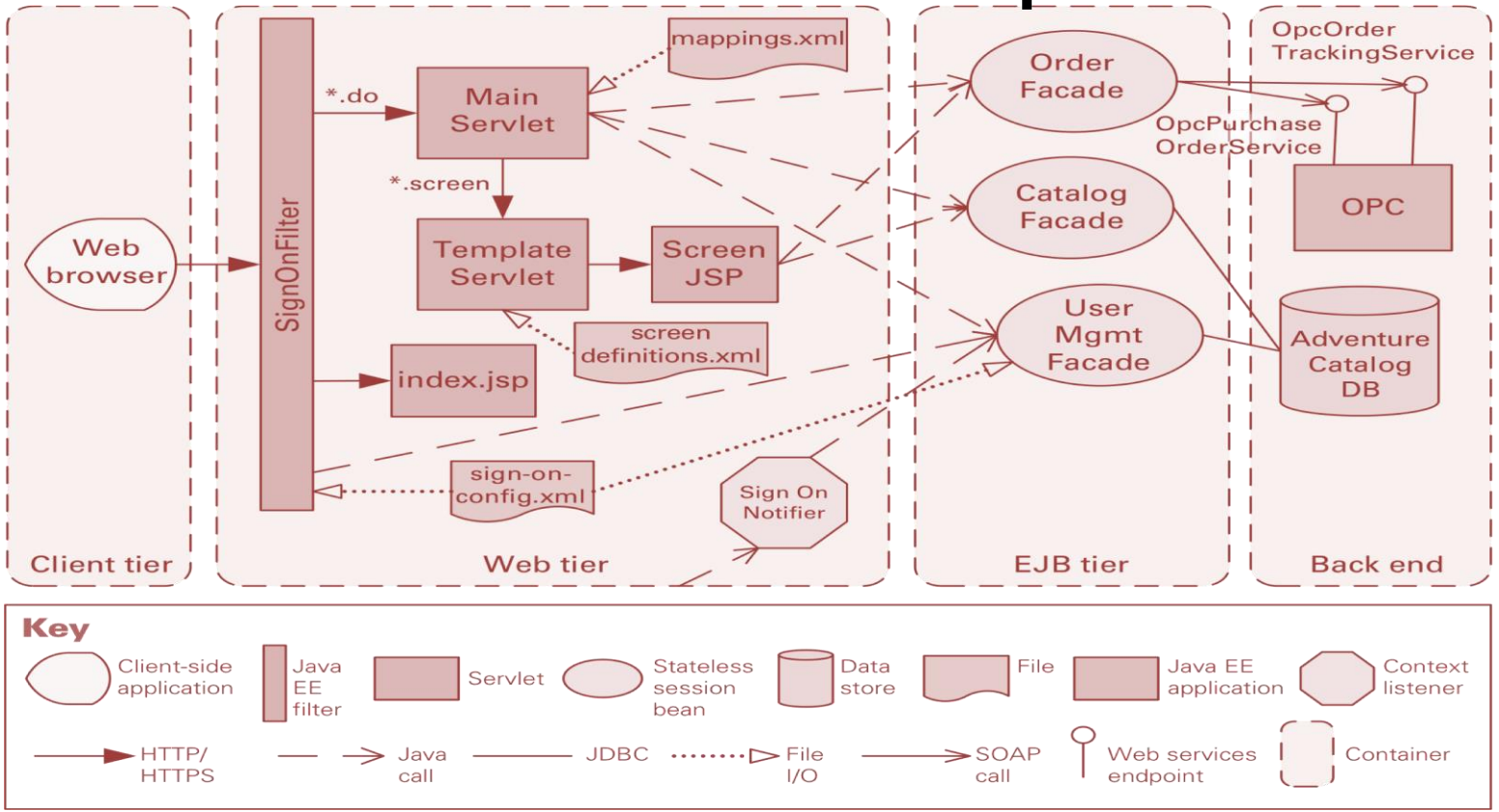
Relations:

- Is part of, to group components into tiers
- Communicates with, to show how tiers and the components they contain interact with each other
- Allocated to, in the case that tiers map to computing platforms

Constraints: A software component belongs to exactly one tier

Weaknesses: Substantial up-front cost and complexity

Multi-Tier Example



References

UNIVERSITY OF CANBERRA

- Len, Bass, Clements Paul, and Kazman Rick. (2013) "Software architecture in practice." Boston, Massachusetts Addison. 3rd Edition.
  - CHAPTER 13 - Architectural Tactics and Patterns