Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Styles and patterns

and their relation to:
viewpoints,
perspectives
and
quality properties

A quick recap

- Views:
  - Address some concern(s) of (a) stakeholder(s)
  - Organized in viewpoints
  - Lot's of freedom!
- We already saw > 20 possible views!
  - How to choose?
  - How to organize?
  - What to put in?
- How to handle this freedom?

Chapter Outline

- What is a Pattern?
- Pattern Catalogue
  - Module patterns
  - Component and Connector Patterns
  - Allocation Patterns

A pattern

- "any regularly repeated arrangement, especially a design made from repeated lines, shapes, or colours on a surface"
- "a particular way in which something is done, is organized, or happens"
- "a drawing or shape used to show how to make something"
- "something that is used as an example, especially to copy"

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: A successful architectural resolution to the problem, appropriately abstracted
- Consequences: results and trade-offs of the pattern

Solutions for a pattern

- A set of element types
  - Data repositories,
  - Processes,
  - Objects
- A set of interaction mechanisms or connectors
  - method calls,
  - events,
  - message bus
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Why patterns?

- Design reuse
  - Well-understood solutions applied to new problems
- Code reuse
  - Shared implementations of invariant aspects of a style
- Understandability of system organization
  - A phrase such as "client-server" conveys a lot of information
- Interoperability
  - Supported by style standardization
- Style-specific analyses
  - Enabled by the constrained design space
- Visualizations
  - Style-specific depictions matching engineers' mental models

Pattern analysis dimensions

- What is the design vocabulary?
  - Component and connector types
- What is the underlying computational model?
- What are the essential invariants of the pattern?
- What are common examples of its use?
- What are the (dis)advantages of using the pattern?
- What are the style's specializations?

Patterns vs styles

- Architectural style: fundamental structural organization schema for software systems
- Pattern: documents commonly recurring and proven structure
- Think of it as in language idiom: idiom describes the patterns of the language.

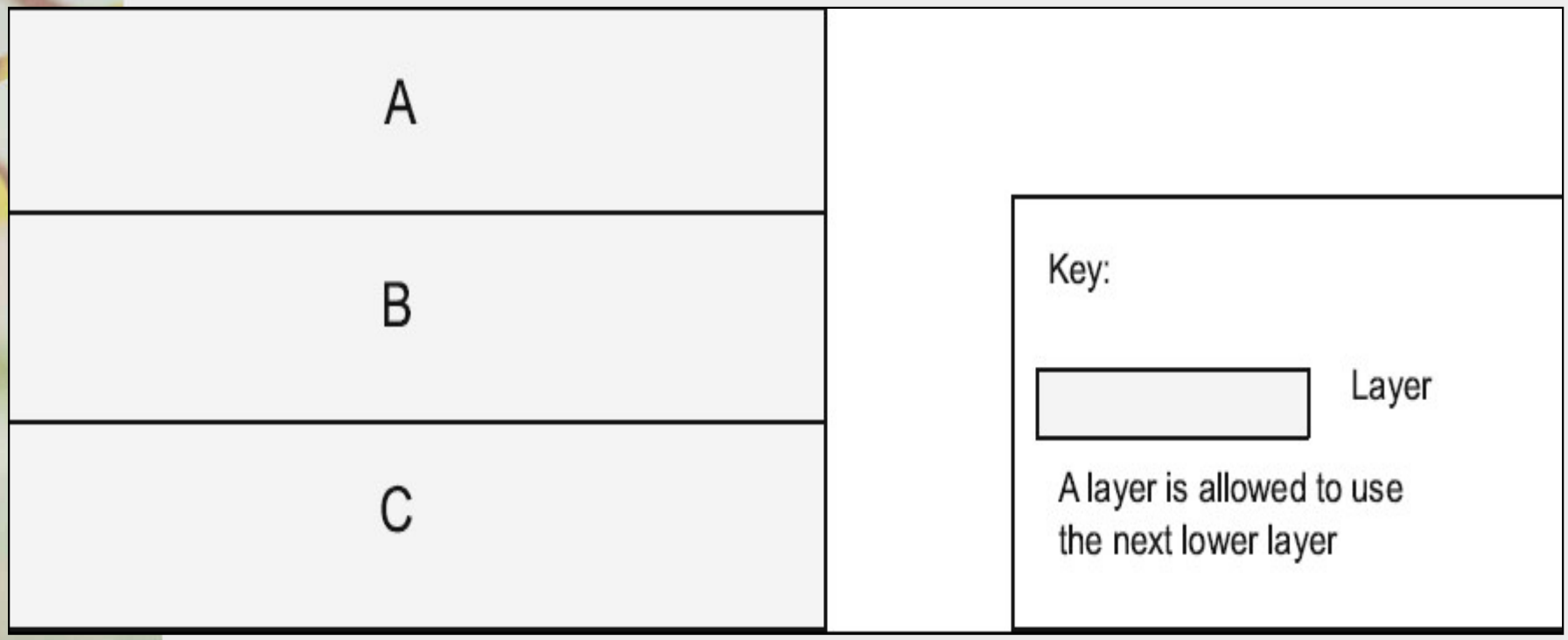
Examples of patterns

- Layer
- Broker
- Model-view-controller
- Pipe-and-filter
- Client-server
- Peer-to-peer
- Service oriented architecture
- Publish-subscribe
- Shared-data
- Multi-tier
- …

Pattern template

- Context
- Problem
- Solution
  - Element types
  - Interaction mechanisms (connectors)
  - Topological layout
  - Constraints

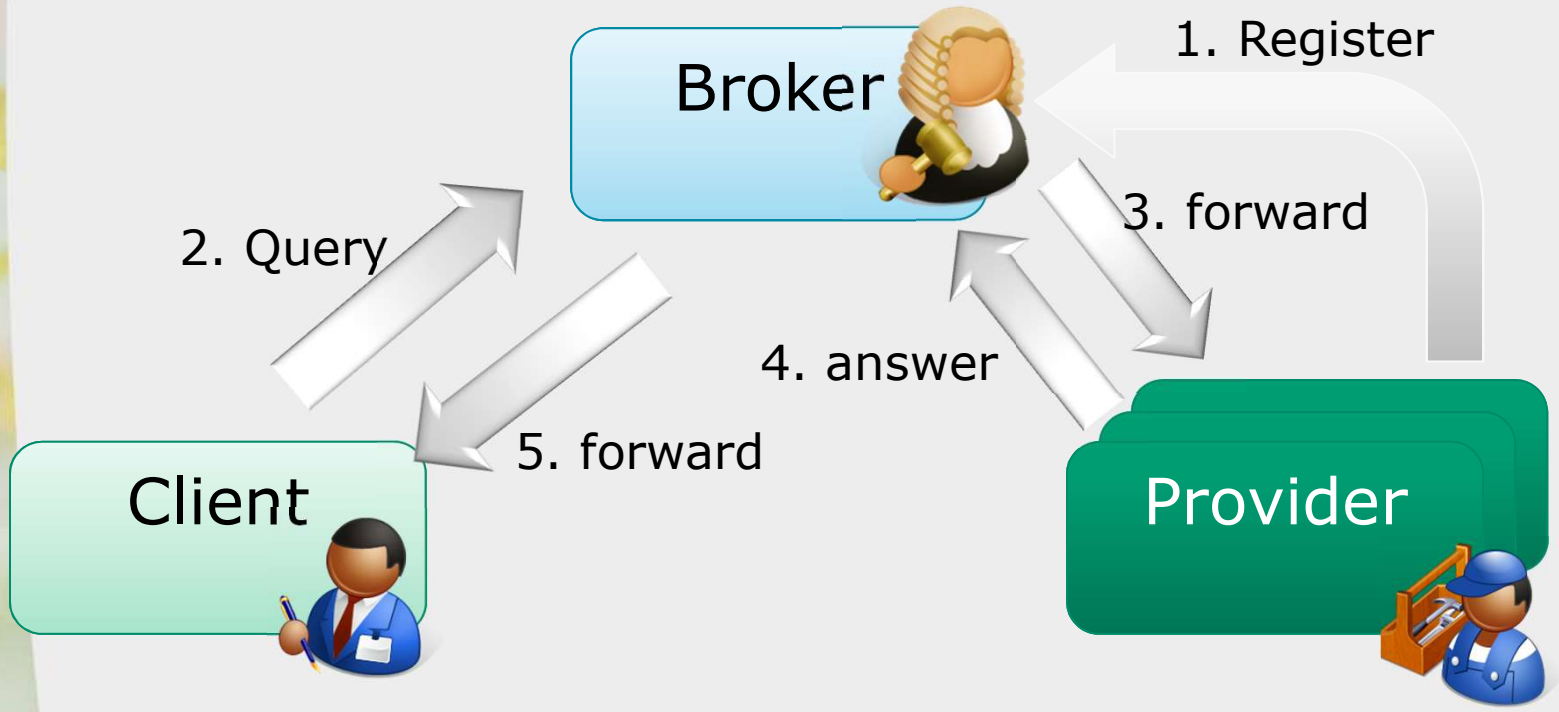
Layer pattern: topoligical layout



Layer pattern

- Context:
  - Develop and evolve portions of the system independently
  - Clear and well-documented separation of concerns
  - Modules independently developed and maintained
- Problem:
  - Segment software to be developed and evolved separately
  - Support portability, modifiability, and reuse
- Solution:
  - Element types:
    - Layer: grouping of modules that offers a cohesive set of services
  - Connectors:
    - public interface
  - Constraints:
    - unidirectional

Broker pattern



Broker pattern

- Mediates communication between clients and servers
- Elements:
  - Client requests a service
  - Server provides a service
  - Broker mediates between clients and services
- Connectors:
  - Attachment: client → broker, server → broker
- Constraints
  - No client → server or server → client

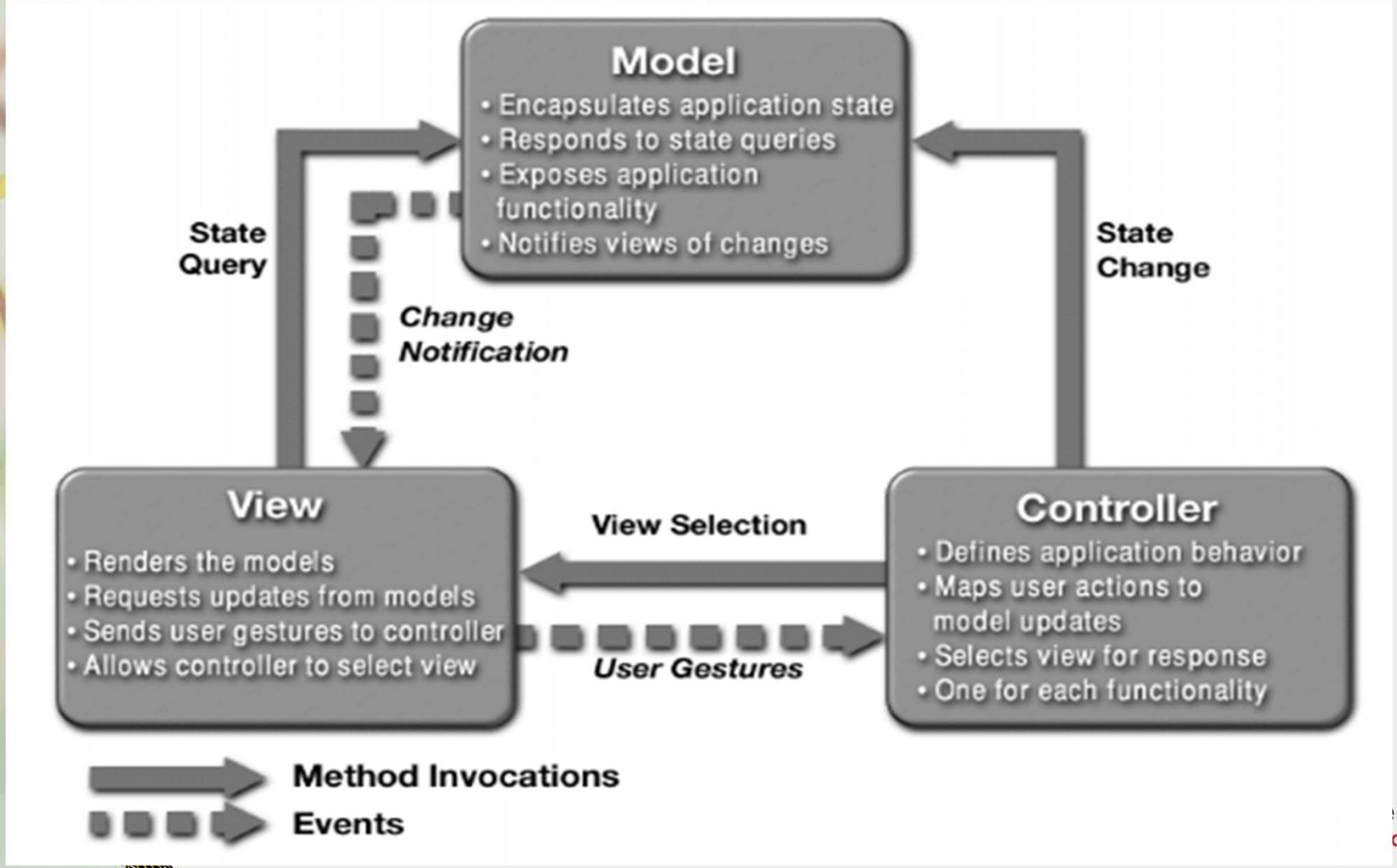
Broker pattern

- Context:
  - Collection of services distributed across multiple servers
  - Availability of these services?
  - How do they interoperate (discover, connect & exchange)?
- Problem:
  - How to structure distributed software
  - Users should not need to know nature and location of service
  - Easy to dynamically change bindings between users and providers
- Solution:
  - Element types:
    - Server, Client, Broker
  - Connectors:
    - Service interface broker
  - Constraints:
    - Only broker knows providers

Broker pattern: weaknesses

- Brokers add a layer of indirection,
  - Latency between clients and servers,
  - Broker may be a communication bottleneck
- The broker can be a single point of failure.
- Broker adds up-front complexity.
- Broker may be a target for security attacks.
- Broker may be difficult to test.

Model-View-Controller pattern



Model-View-Controller pattern

- Context:
  - User interface most frequently updated part of software
  - Data displayed from different representations
  - Representations should all reflect current state of the data
- Problem:
  - Keep UI functionality separate from application functionality
  - Keep responsive to both data changes and user input
  - Create, maintain and coordinate views of the UI consistently
- Solution:
  - Element types:
    - Model, view, controller
  - Connectors:
    - Notify
  - Constraints:
    - Model does not communicate with view directly

Model-View-Controller solution

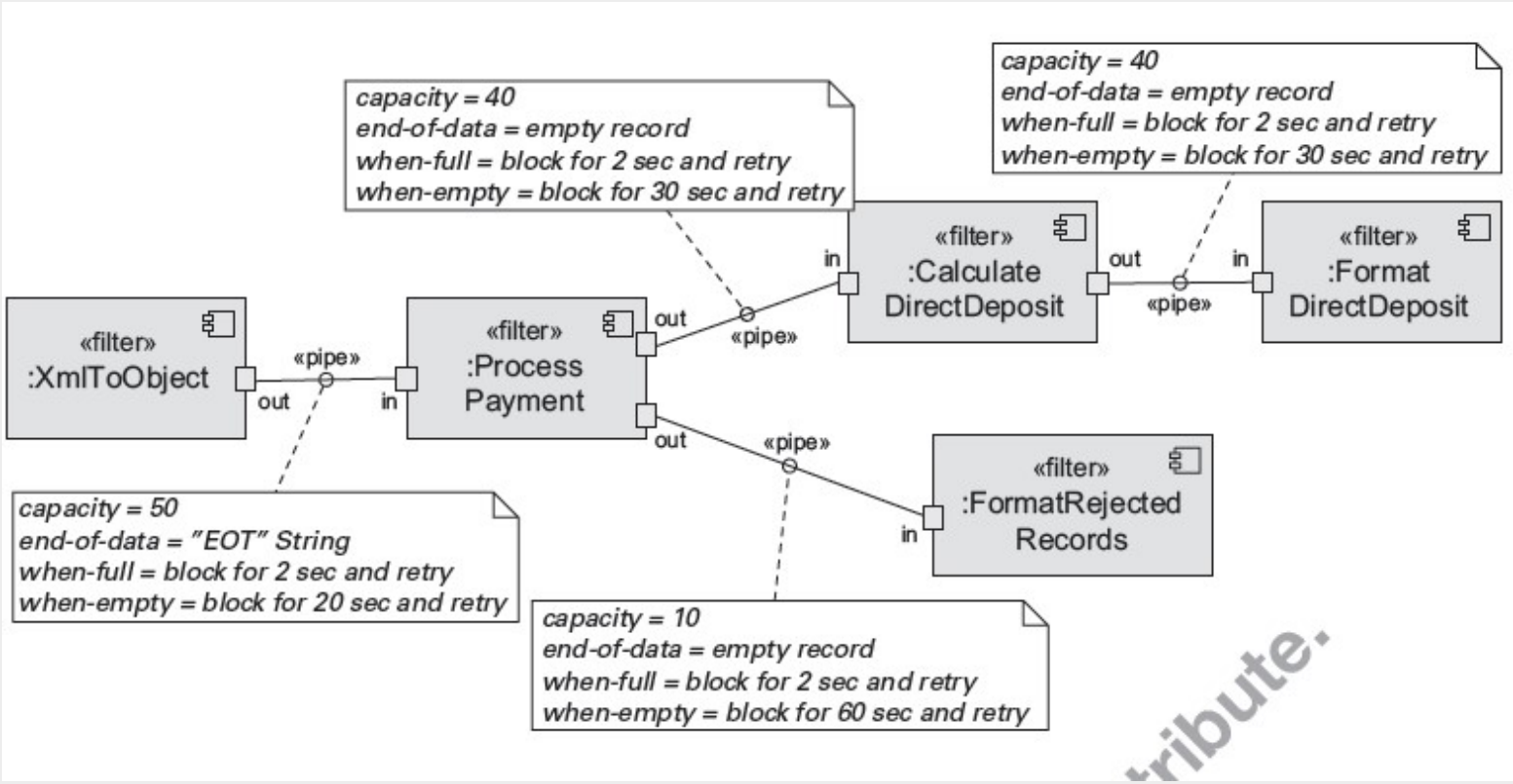
Elements:

- Model:
  - Representation of the application data or state,
  - Contains (or provides an interface to) application logic.
- View:
  - User interface component
  - Produces a representation of the model for the user
  - Allows for some form of user input
- Controller:
  - Manages all interaction between model and view
  - Translates actions into changes to the model or to the view.

Model-View-Controller solution (2)

- Constraints:
  - At least one instance of model, view, and controller
  - Model component should not interact directly with controller
- Weaknesses:
  - High complexity
  - Abstractions may not be good fits for some UI toolkits

Pipe-and-filter pattern



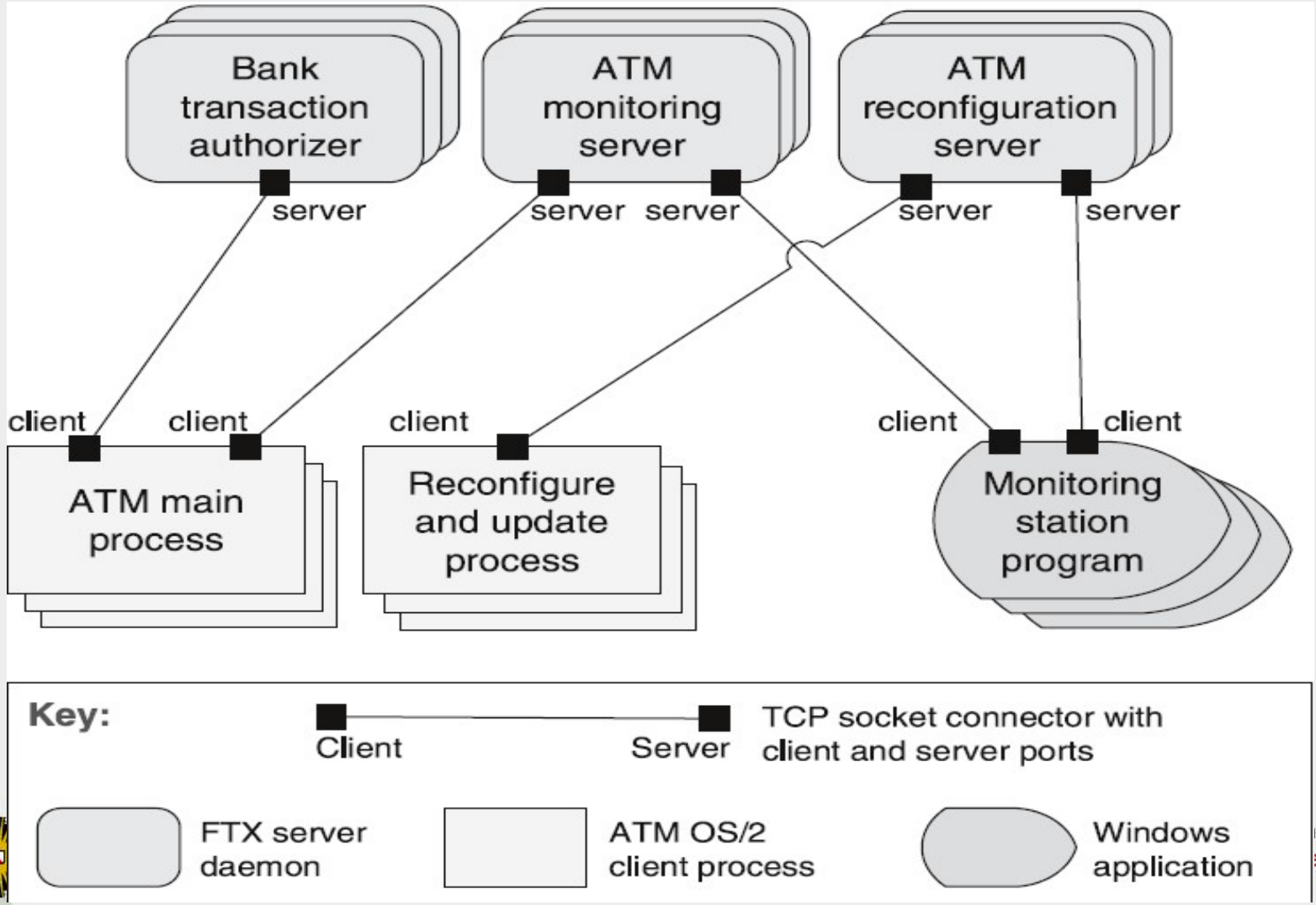
Pipe-and-filter pattern

- Context:
  - Many systems needed to transform streams of discrete data items, from input to output.
  - Transformations occur repeatedly in practice
- Problem:
  - Divide into reusable, loosely coupled components
  - Simple, generic interaction mechanisms
- Solution:
  - Element types:
    - Filter
  - Connectors:
    - Pipe, input-output interface
  - Constraints:
    - Input-output interface filters should match

Pipe-and-filter solution

- A series of transformations performed by filters connected by pipes
- Elements
  - Filter: transforms data read on input port(s) to data on its output port(s)
  - Pipe: conveys data from filter's output port to another filter's input port
- Connectors
  - Association: connect output port of filters to input port of filters
- Constraints
  - Pipes connect filter output ports to filter input ports
  - Connected filters must agree on the type of data being passed along the connecting pipe

Client-server pattern



Client-server pattern

- Context:
  - Shared resources
  - Large numbers of distributed clients,
  - Control access of clients or quality of service
- Problem:
  - Improve scalability and availability
  - Distribute resources across multiple physical servers
- Solution:
  - Element types:
    - Client, Server
  - Connectors:
    - Request/Reply
  - Constraints:

Client-server solution

Elements:

- Client:
  - Invokes services of a server component.
  - Clients have ports that describe the services they require.
- Server:
  - Provides services to clients.
  - Servers have ports that describe the services they provide.

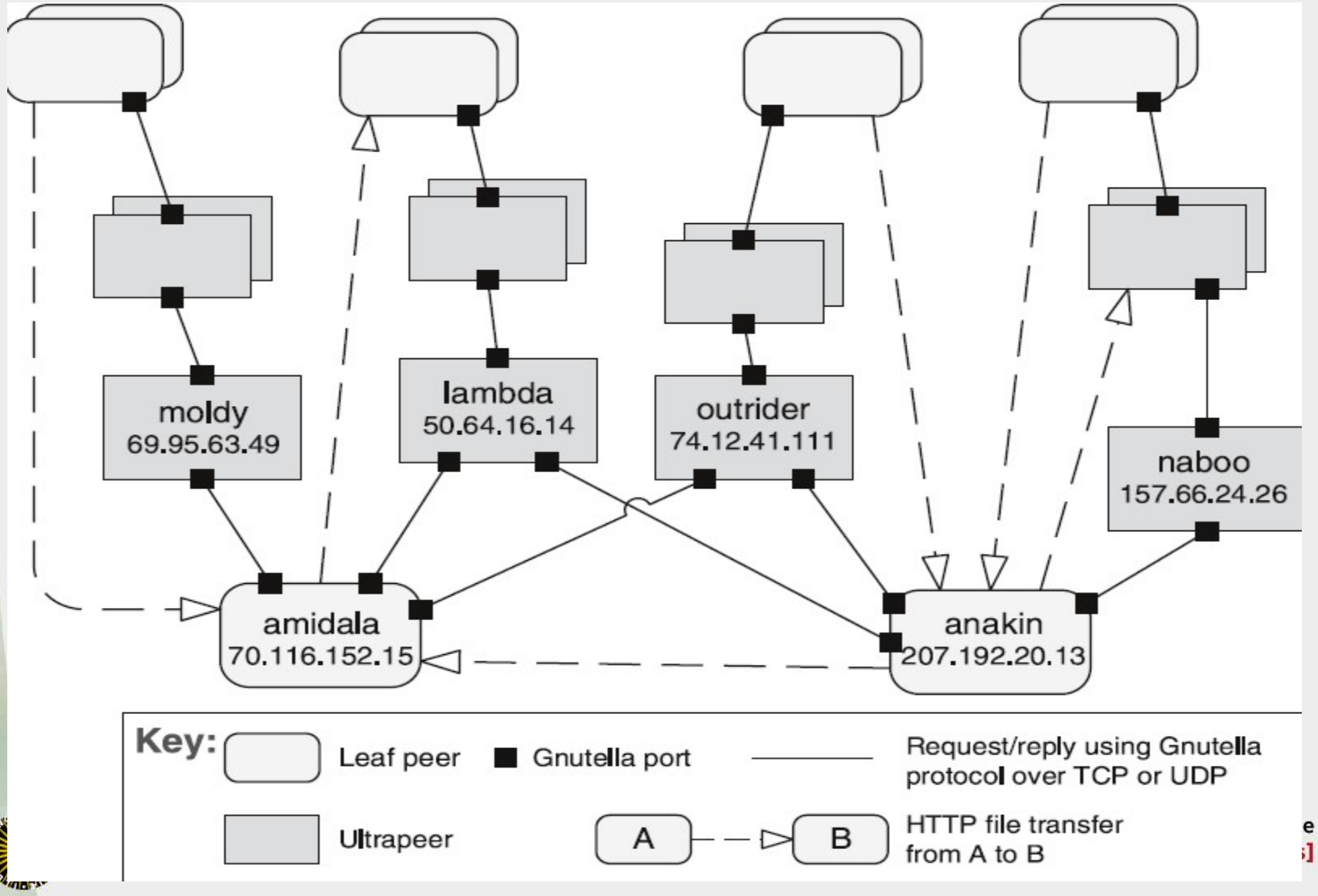
Connectors:

- Request/reply:
  - Employs a request/reply protocol
  - Calls are local or remote
  - data is encrypted or not.

Client-server solution

- 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
    - Complex
    - Costly to change after a system has been built

Peer-to-peer pattern



Peer-to-peer pattern

- Context:
  - Distributed computational, equally important entities
  - Cooperate and collaborate to provide a service to a distributed community of users.
- Problem:
  - How to connect "equal" distributed computational entities?
  - How to organize and share their services with high availability and scalability?
- Solution:
  - Element types:
    - Peer
  - Connectors:
    - Request/Reply
  - Constraints:

Peer-to-peer solution

- Computation is achieved by cooperating peers that request service from and provide services to one another across a network

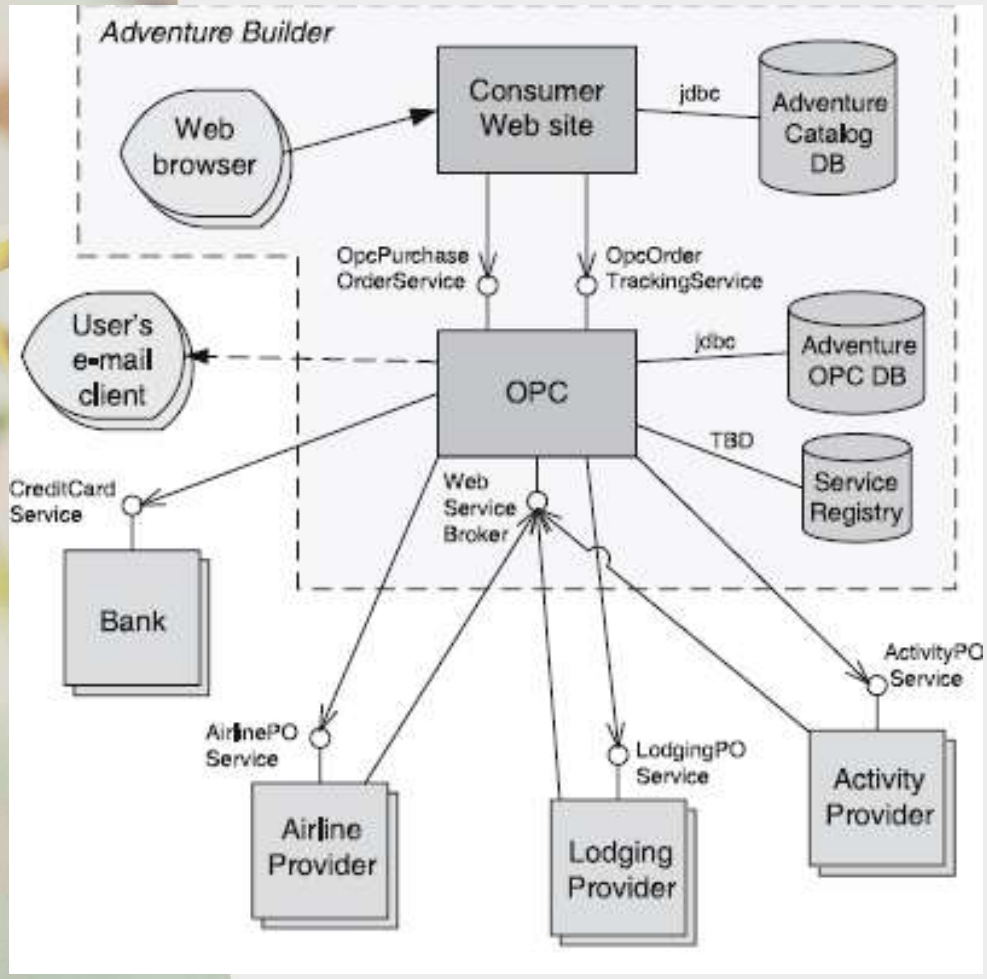
Elements:

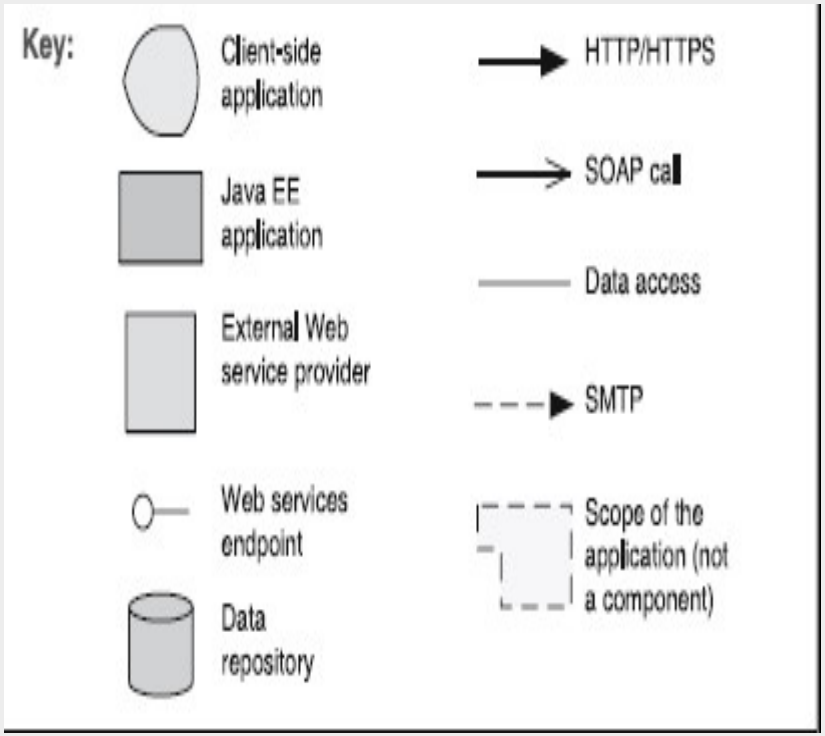
- Peer:
  - Independent component running on a network node.
  - Often provide routing, indexing, and peer search capability.
- Request/reply connector:
  - Connect to peer network
  - Search for other peers,
  - Invoke services from other peers

Peer-to-peer solution

- Relations / network:
  - Star network
  - Supernodes
  - Neighbours only
- Constraints:
  - 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
- 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

Service Oriented Architecture pattern





Service Oriented Architecture pattern

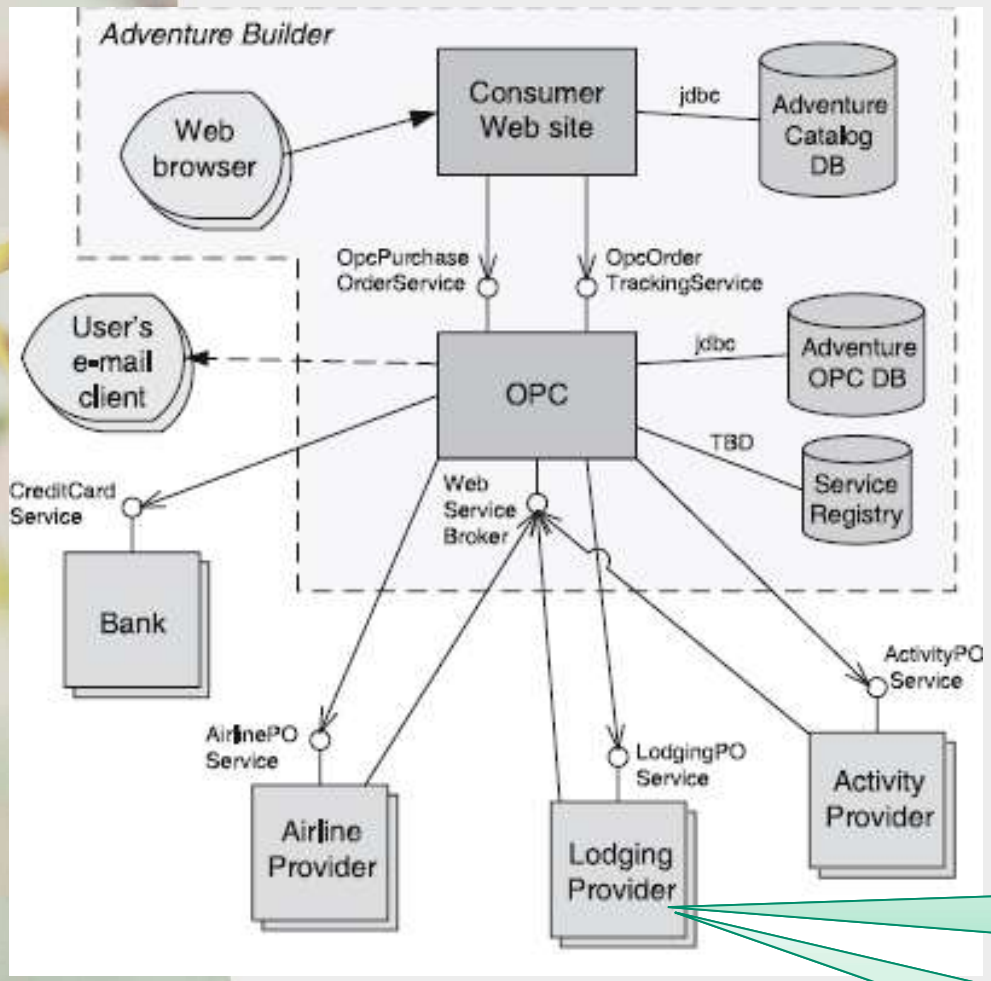
- Context:
  - Consumers need to be able to understand and use providers without any detailed knowledge of their implementation
- Problem:
  - How to support interoperability of distributed components? Provided by different organizations, distributed across Internet?
- Solution:
  - Element types:
    - Server, client
  - Connectors:
    - Interface usage / offering
  - Constraints:

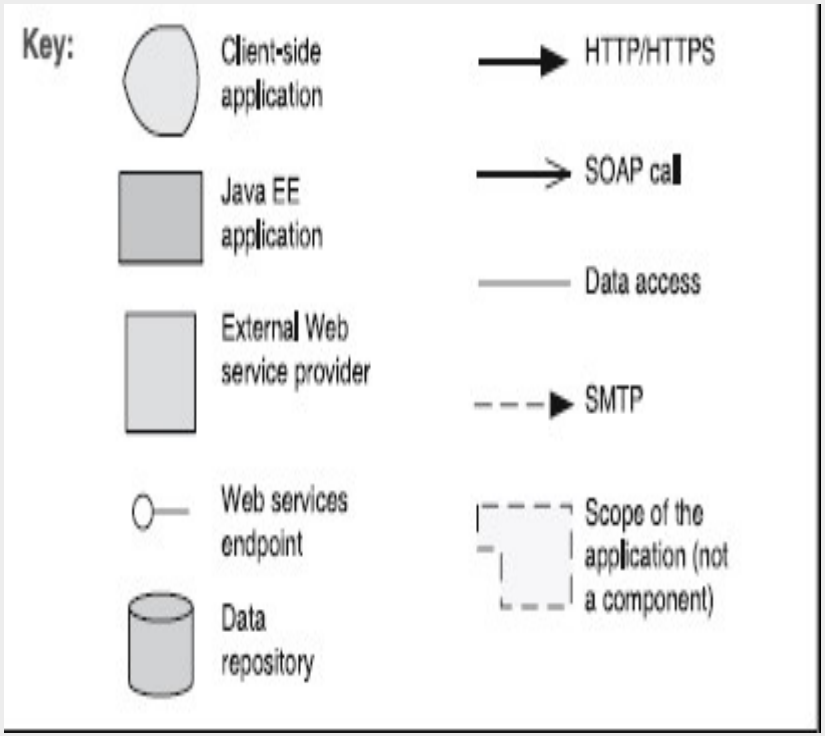
Service Oriented Architecture solution

Elements:

- Component:
  - Service provider: offers a service
  - Service consumer: invokes a service
- Enterprise Service Bus:
  - Route and transform messages between service providers and consumers
- Registry:
  - Knows the services providers offer
- Orchestrator:
  - Coordinates the interactions between service consumers and providers

Service Oriented Architecture pattern



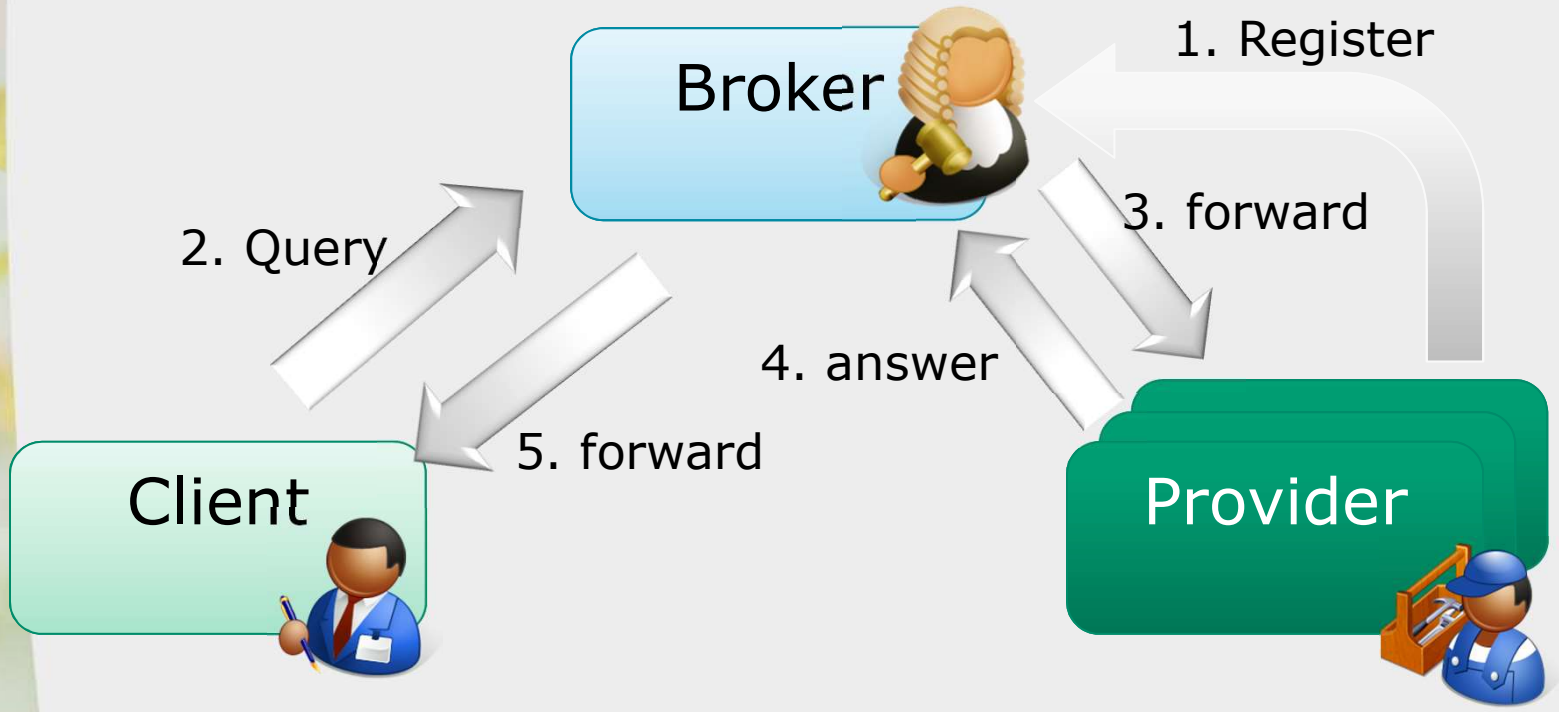


Where are these services located?

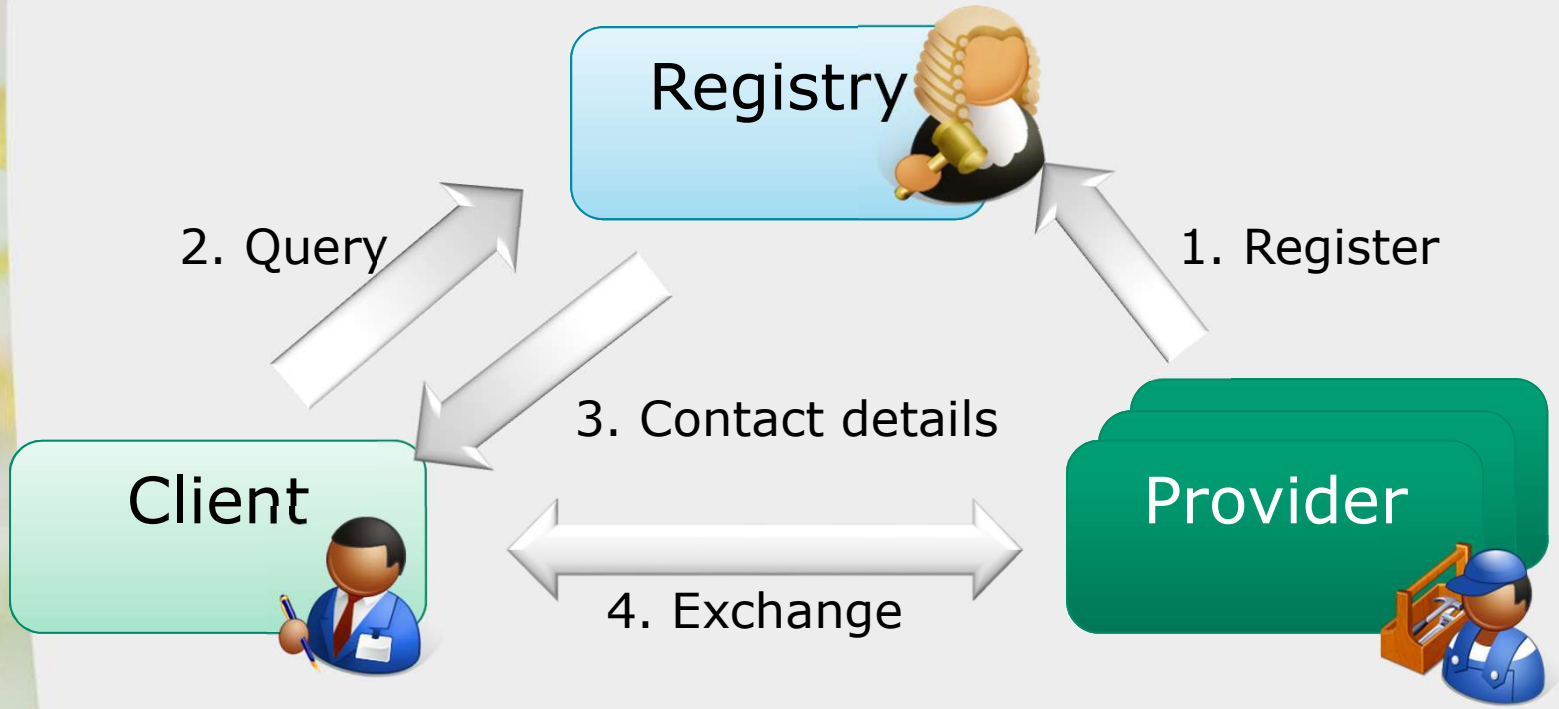
How to find these services?

Universiteit Utrecht

Broker pattern



Registry in SOA



Service Oriented Architecture solution

Connectors:

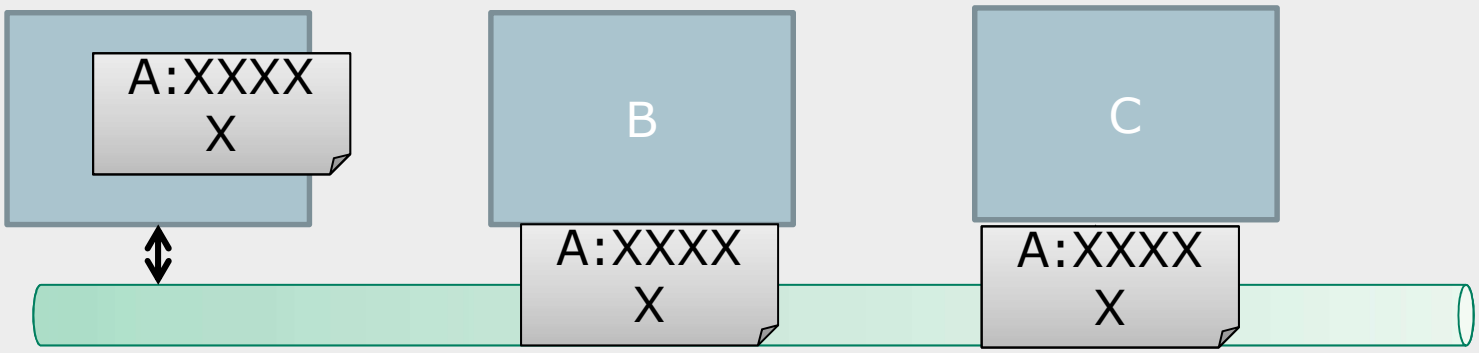
- SOAP connector
  - Uses the SOAP protocol for communication between services, can be over HTTP, SMTP, …
- REST connector
  - relies on the basic request/reply operations of the HTTP protocol.
- Asynchronous messaging connector,
  - Uses a messaging system

Service Oriented Architecture solution

- Weaknesses:
  - SOA-based systems are typically complex to build.
  - You don't control the evolution of independent services.
  - Performance overhead with middleware,
  - Services may be performance bottlenecks,
  - No performance guarantees

Publish-Subscribe solution

- Elements
  - Publisher: writes on a channel
  - Subscriber: reads from a channel
  - Channel: distributes events between publishers and subcribers



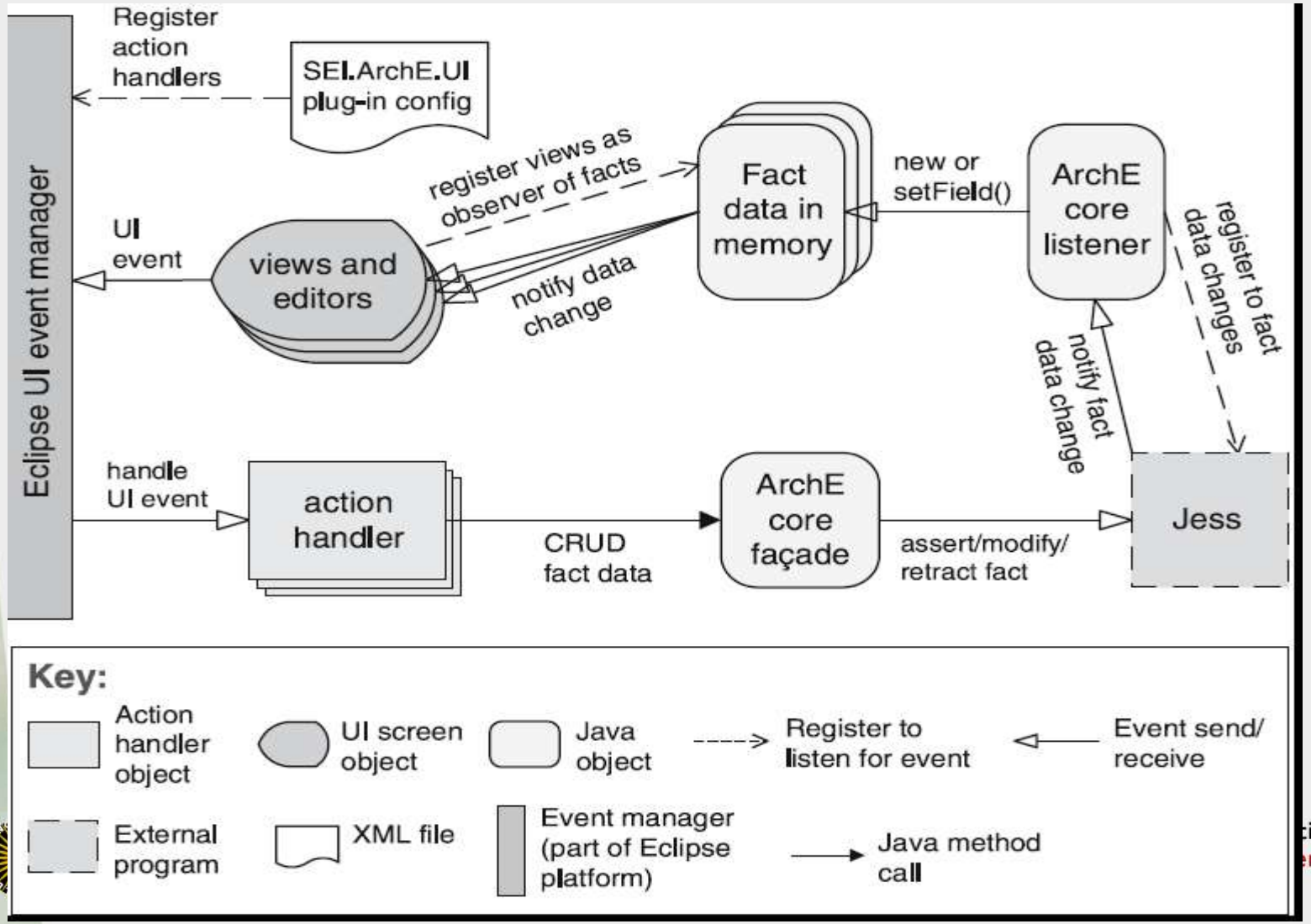
Publish-Subscribe pattern

- Context:
  - Independent producers and consumers of data must interact
  - Nature of data not predetermined or fixed
- Problem:
  - Integration mechanisms that support the ability to transmit messages between producers and consumers that are unaware of each other
- Solution:
  - Element types:
    - Publisher, Subcriber, Blackboard, Event
  - Connectors:
    - Channel
  - Constraints:

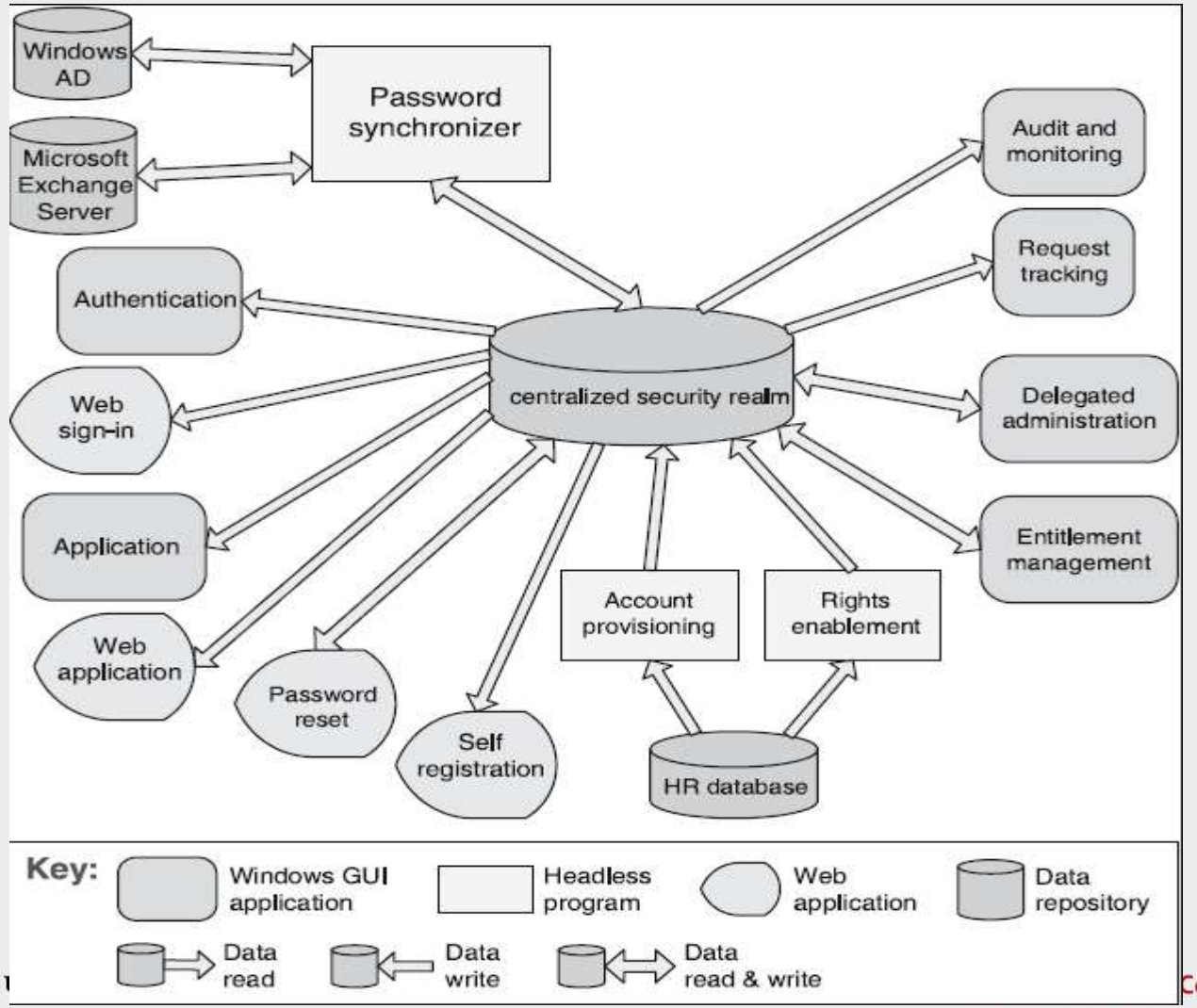
Publish-Subscribe solution

- Constraints:
  - All components are connected to an event distributor
- Weaknesses:
  - Increases latency
  - Negative effect on scalability
  - Negative effect on predictability of message delivery time
  - Less control over ordering of messages
  - Delivery of messages is not guaranteed

Publish-Subscribe pattern



Shared-data pattern



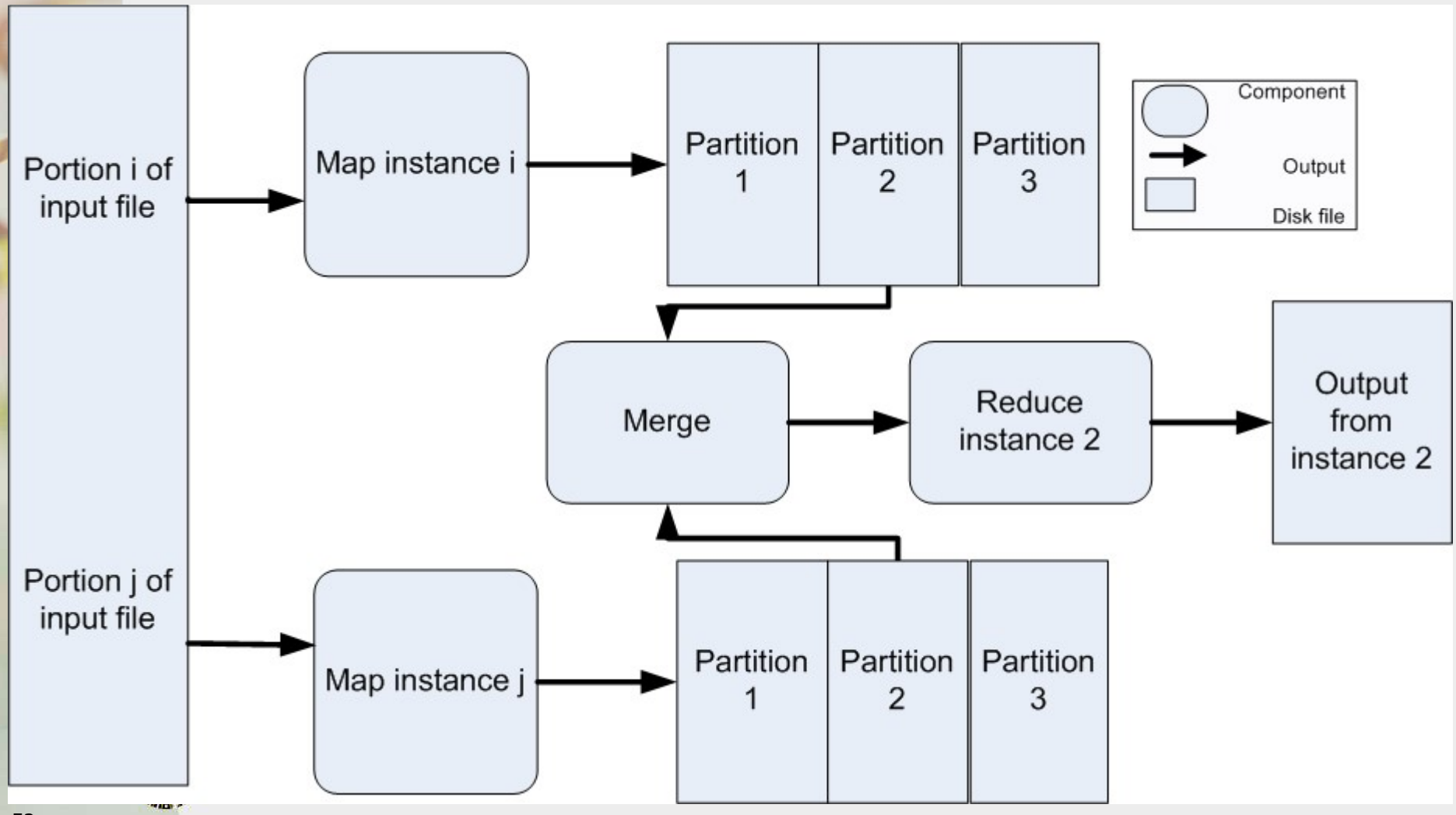
Shared-data pattern

- Context:
  - Components share and manipulate large amounts of data
  - No component owner of the data
- Problem:
  - How can systems store and manipulate persistent data that is accessed by multiple independent components?
- Solution:
  - Element types:
    - Accessor, Shated data store
  - Connectors:
    - Data accessor
  - Constraints:

Shared-data solution

- Elements:
  - Shared data store: manages access control and data persistency
  - Accessor:
- Weaknesses:
  - Shared-data store may be a performance bottleneck
  - Shared-data store may be a single point of failure
  - Producers and consumers of data may be tightly coupled

Map-Reduce pattern



Map-Reduce pattern

- Context:
  - Quickly analyze enormous volumes of data
- Problem:
  - Ultra-large data sets
  - Efficiently perform a distributed and parallel sort
- Solution:
  - A specialized infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and handles sorting the data as needed
  - A programmer specified component called the map which filters the data to retrieve those items to be combined
  - A programmer specified component called reduce which combines the results of the map

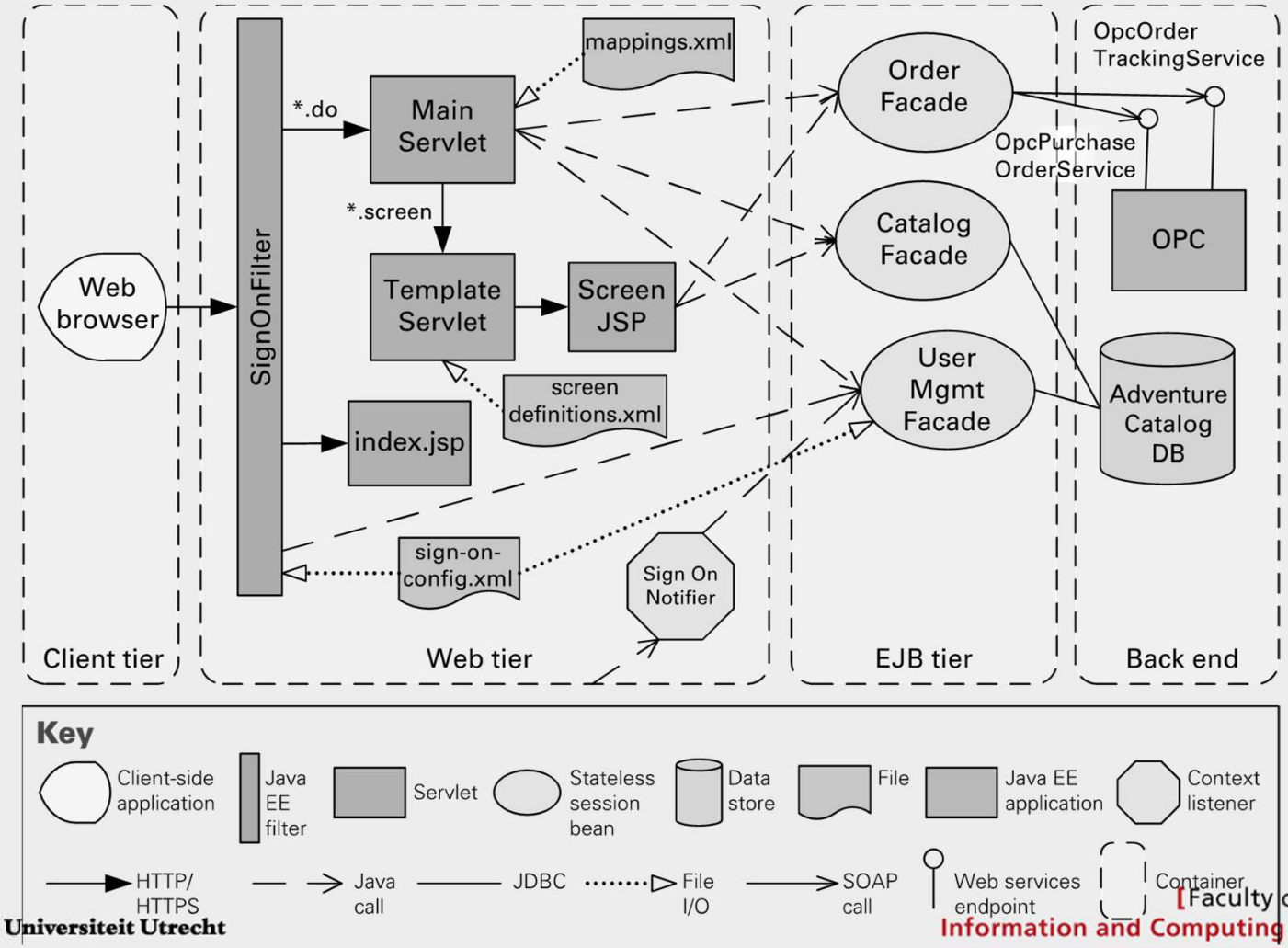
Map-Reduce pattern

- Elements:
  - Map is a function with multiple instances deployed across multiple processors that performs the 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: relation between map or reduce function and processor
  - Instantiate, monitor, and control
- Constraints:
  - The data to be analyzed must exist as a set of files
  - Map functions are stateless and do not communicate
  - 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 requiring multiple reduces are complex to orchestrate

Multi-Tier pattern



Multi-Tier pattern

- Context:
  - Distribute a system's infrastructure into distinct subsets
- Problem:
  - Split system in computationally independent execution structures
- Solution:
  - Element types:
    - Tier: logical grouping of components
  - Connectors:
    - Part of, communicates-with, allocated-to
  - Constraints:
    - Communication only with tier above and below
- Weakness:
  - Substantial up-front cost and complexity

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

To summarize

To summarize

- Many patterns exist
  - Context – what is the environment of our problem
  - Problem – what do I want to achieve? Can I generalize?
  - Solution – what is a possible solution?
- Solution:
  - Element types
  - Interaction mechanisms
  - Topological layout
  - Constraints
- Patterns are one of the first design decisions you will make!
- Which one suits better?
  - Use rationale to weight pros and cons