Documentation for Software Architecture Patterns

---

1. Client-Server Pattern

Overview

The client-server pattern structures systems to provide shared services to multiple clients, centralizing the management of resources and services.

Context

Used when multiple clients need access to shared resources and services, aiming to improve modifiability, scalability, and centralized control.

Problem

The challenge is to centralize resource management while distributing resources across multiple physical servers, ensuring easier modification, reuse, scalability, and availability.

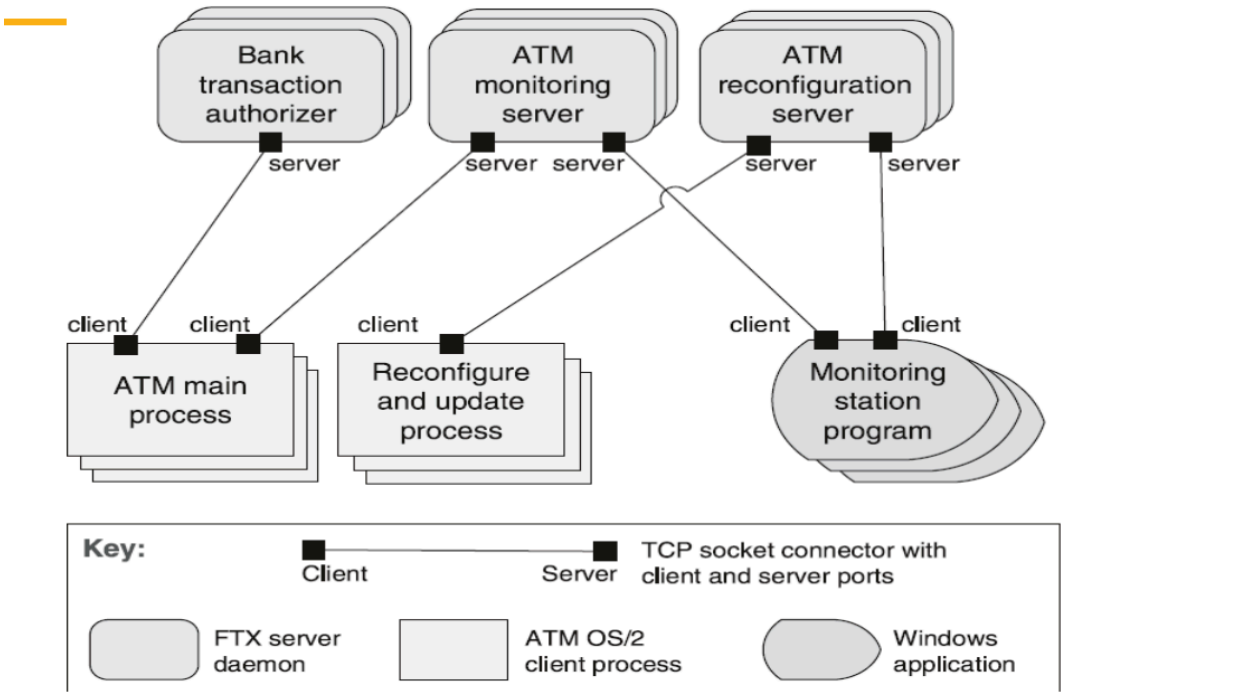
Solution

- **Clients** request services.
- **Servers** provide services.
- Clients interact with servers via request/reply protocols.
- Components can act as both clients and servers.

Use Case Example

In a banking system, the server manages customer accounts, while clients (e.g., web apps, ATMs) send requests to access or modify account data.

Diagram: Client-Server Example



---

2. Peer-to-Peer (P2P) Pattern

Overview

In the peer-to-peer pattern, components (peers) directly interact with each other to provide distributed services.

Context

Used in systems where distributed computational entities need to collaborate equally, without a central controlling component.

Problem

How to enable a set of distributed, "equal" computational entities to connect, share, and organize services while maintaining high availability and scalability.

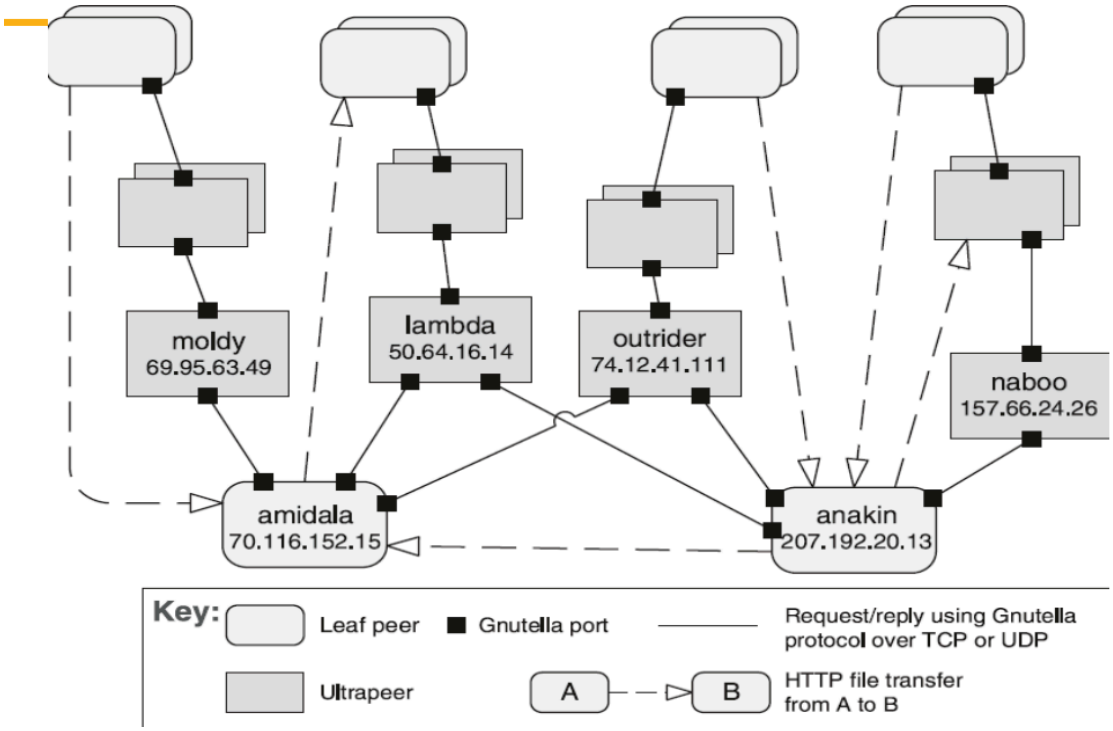
Solution

- **Peers** interact directly as equals, requesting and providing services.
- **Request/reply connectors** are used to search for and communicate with other peers.

Use Case Example

In a file-sharing system, users can both share (provide) and download (request) files from other users' devices.

Diagram: Peer-to-Peer Example



---

3. Publish-Subscribe Pattern

Overview

The publish-subscribe pattern enables components to interact by publishing events that are distributed to subscribed components.

Context

Useful when the number and nature of data producers and consumers are variable and unpredictable.

Problem

How to enable data integration among independent producers and consumers without knowing each other's identity or existence.

Solution

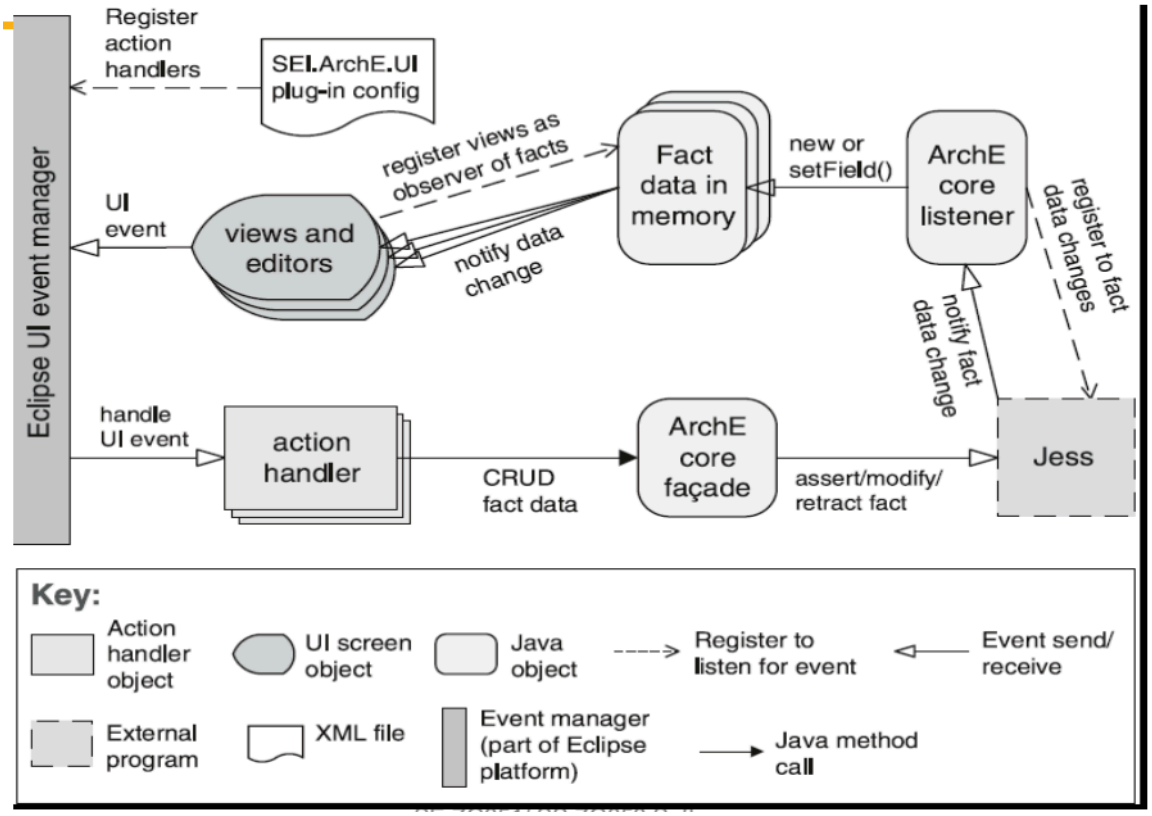
- **Publishers** announce events.

- **Subscribers** listen for events.
- The **publish-subscribe connector** delivers events from publishers to subscribers.

Use Case Example

In a news delivery system, different channels publish news updates, and users subscribe to channels of interest to receive updates.

Diagram: Publish-Subscribe Example



---

4. Shared-Data Pattern

Overview

The shared-data pattern is used when multiple independent components need to access and manipulate large, persistent data.

Context

Applied when persistent data needs to be shared among several components without being exclusive to any one of them.

Problem

How to manage persistent data that needs to be accessed by multiple, independent components.

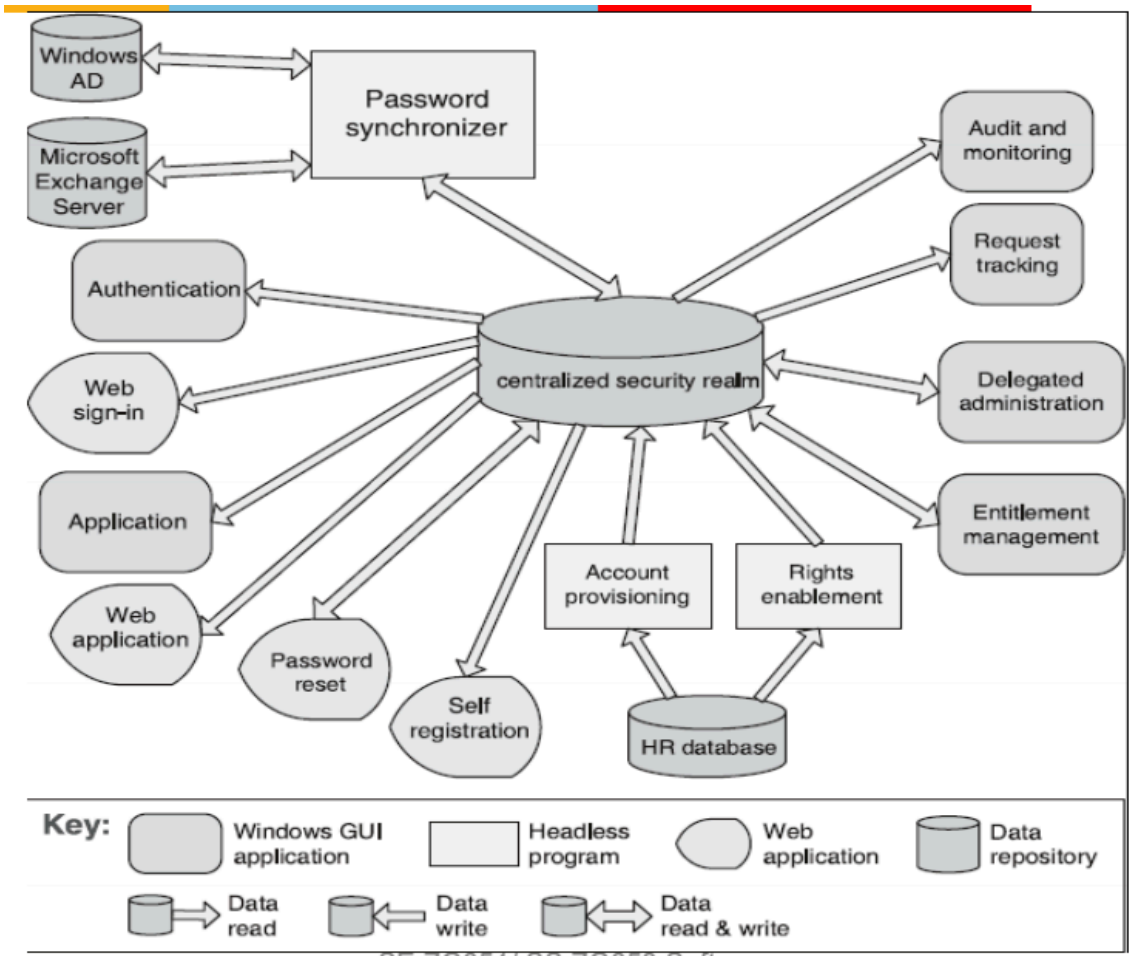
Solution

- A **shared-data store** maintains persistent data.
- **Data accessors** read from and write to the shared-data store.

Use Case Example

In a hospital management system, the shared database holds patient records, which are accessed by doctors, nurses, and administrative staff.

Diagram: Shared-Data Example



Additional Details for Each Pattern

1. Client-Server Pattern

- **Elements:**
  - **Client**: Invokes services of servers.
  - **Server**: Provides services to clients.
  - **Request/reply connector**: Manages client-server communication.
- **Relations**: Clients connect to servers via request/reply connectors.
- **Constraints**:
  - Servers may act as clients to other servers.
- **Weaknesses**:
  - Servers can become performance bottlenecks or single points of failure.

2. Peer-to-Peer (P2P) Pattern

- **Elements:**
  - **Peer**: A component that both requests and provides services.
  - **Request/reply connector**: Handles peer communication.
- **Relations**: Peers are connected via request/reply connectors.
- **Constraints**:
  - Number of connections and search depth may be limited.
- **Weaknesses**:
  - Complexity in managing security, data consistency, and availability.

3. Publish-Subscribe Pattern

- **Elements:**
  - **Publisher**: Publishes events.
  - **Subscriber**: Listens for events.
  - **Publish-subscribe connector**: Distributes events.
- **Relations**: Components connect via the publish-subscribe connector.
- **Constraints**: All components are linked to an event distributor.
- **Weaknesses**:
  - Can increase latency, reduce scalability, and create unpredictable message delivery.

4. Shared-Data Pattern

- **Elements:**
  - **Shared-data store**: Holds persistent data.
  - **Data accessor**: Reads and writes data.
  - **Data reading/writing connector**: Manages data exchange.
- **Relations**: Accessors connect to data stores via data connectors.
- **Constraints**: Data accessors interact only with the shared-data store.
- **Weaknesses**:
  - Performance bottlenecks and single points of failure.