Curs 13

Distributed Computing Patterns

Distributed systems

- We may define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only bypassing messages .

- Distributed-Computing is the process of solving a problem using a distributed system.

Characteristics

Concurrency:

- coordination of concurrently executing programs that share resources is also an important and recurring topic.

No global clock:

-there is no single global notion of the correct time.

-This is a direct consequence of the fact that the only communication is by sending messages through a network.

Independent failures:

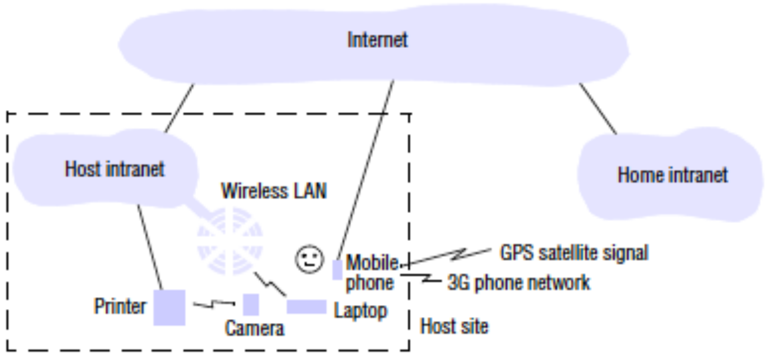
- All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures.
- Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running.
- In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow.
- Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a crash), is not immediately made known to the other components with which it communicates.
- Each component of the system can fail independently, leaving the others still running.

Trends

Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

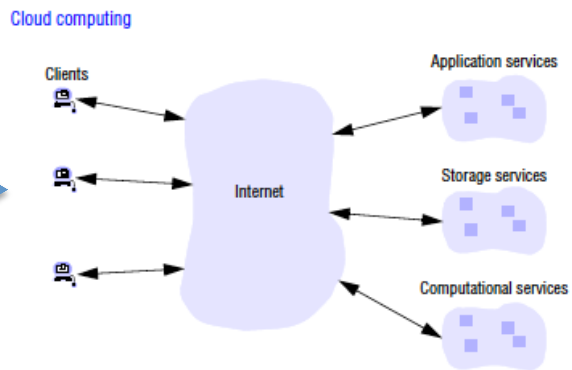
- the emergence of pervasive networking technology;

- the emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;

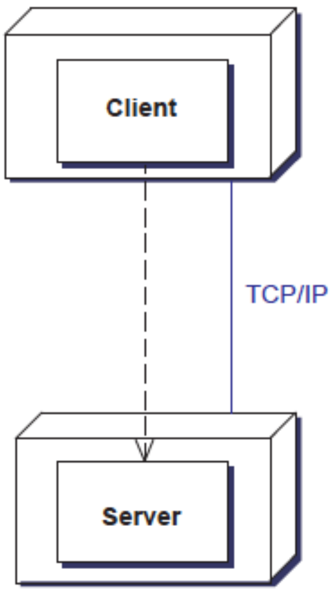


- the increasing demand for multimedia services;

- the view of distributed systems as a utility.



Client-Server



- A server component provides services to multiple client components.
- A client component requests services from the server component.
- Servers are permanently active, listening for clients.

State in the Client-server pattern

Clients and servers are often involved in ‘sessions’.

– With a **stateless server** , the session state is managed by the client. This client state is sent with each request. In a web application, the session state may be stored as URL parameters, in hidden form fields, or by using cookies. This is mandatory for the REST architectural style for web applications.

– With a **stateful server** , the session state is maintained by the server, and is associated with a client-id.

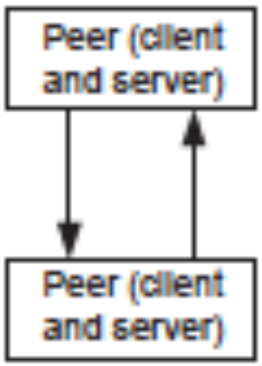
State in the Client-server pattern influences transactions, fault handling and scalability.

- Transactions should be atomic, leave a consistent state, be isolated (not affected by other requests) and durable. These properties are hard to obtain in a distributed world.
- Concerning fault handling, state maintained by the client means for instance that everything will be lost when the client fails.
- Client-maintained state poses security issues as well, because sensitive data must be sent to the server with each request.
- Scalability issues may arise when you handle the server state in-memory: with many clients using the server at the same time, many states have to be stored in memory at the same time as well.

Peer-to-peer pattern

- it can be seen as a symmetric Client-server pattern:
  - peers may function both as a client, requesting services from other peers, and as a server, providing services to other peers.
- A peer may act as a client or as a server or as both, and it may change its role dynamically.
- Both clients and servers in the peer-to-peer pattern are typically multithreaded. The services may be implicit (for instance through the use of a connecting stream) instead of requested by invocation.
- Peers acting as a server may inform peers acting as a client of certain events. Multiple clients may have to be informed, for instance using an event-bus.

Examples



- the distributed search engine Sciencenet,
- multi-user applications like a drawing board,
- peer-to-peer file-sharing like Gnutella or Bittorrent.

Characteristics

- An advantage of the peer-to-peer pattern is that nodes may use the capacity of the whole, while bringing in only their own capacity.
  - In other words, there is a low cost of ownership, through sharing.
- Administrative overhead is low, because peer-to-peer networks are self-organizing.
- The Peer-to-peer pattern is scalable, and resilient to failure of individual peers.
- The configuration of a system may change dynamically: peers may come and go while the system is running.
- A disadvantage may be that there is no guarantee about quality of service, as nodes cooperate voluntarily.
  - For the same reason, security is difficult to guarantee.
- Performance grows when the number of participating nodes grows, which also means that it may be low when there are few nodes.

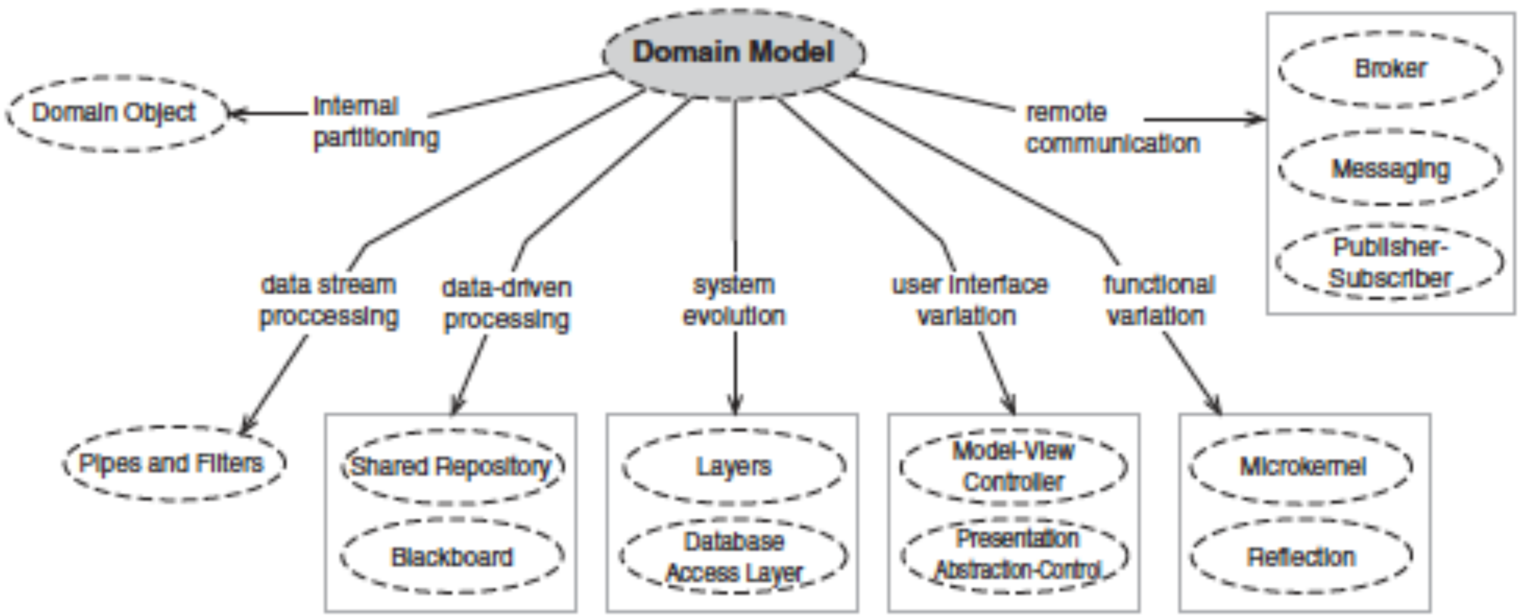
Architectural patterns

Frank Buschmann. Kevlin Henney.Douglas C. Schmidt.**Pattern-Oriented Software Architecture,**Volume 4: A Pattern Language forDistributed-Computing.Wiley & Sons, 2007

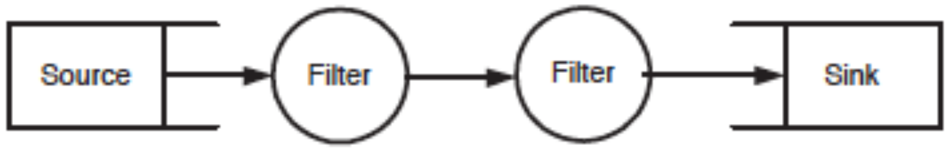
An architectural pattern is a concept that solves and delineates some essential cohesive elements of a software architecture.

- Domain Model
- Layers
- Model-View-Controller
- Presentation-Abstraction-Control
- Microkernel
- Reflection
- Pipes and Filters
- Shared Repository
- Blackboard
- Domain Object

Connection to the Domain Layer

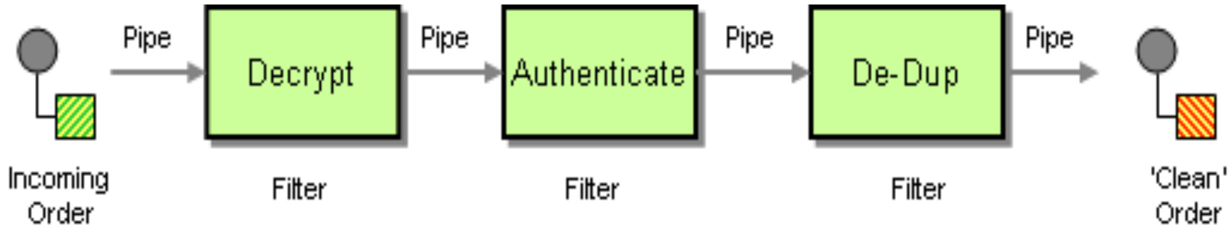


Pipe-Filter Pattern



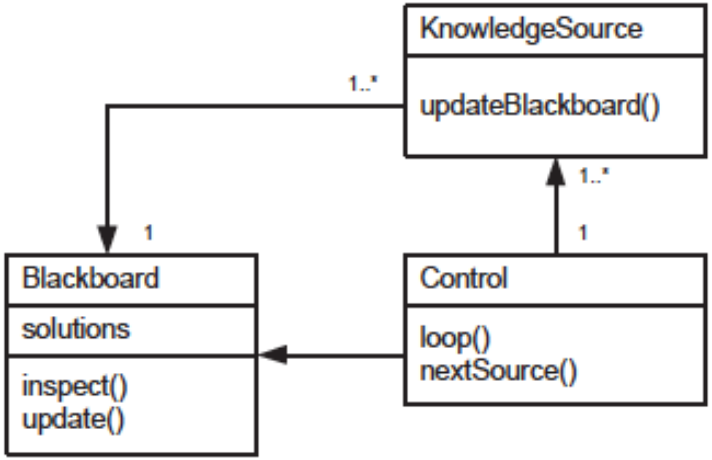
- The Pipe-filter architectural pattern provides a structure for systems that produce a stream of data.
- Each processing step is encapsulated in a filter component (circles).
- Data is passed through pipes (the arrows between adjacent filters).
  - The pipes may be used for buffering or for synchronization.

Example



- Use the Pipes and Filters architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes).
- Each filter exposes a very simple interface: it receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe.
- The pipe connects one filter to the next, sending output messages from one filter to the next.
- Because all component use the same external interface they can be composed into different solutions by connecting the components to different pipes.
- We can add new filters, omit existing ones or rearrange them into a new sequence -- all without having to change the filters themselves.

Blackboard Pattern



- The Blackboard pattern is useful for problems for which no deterministic solution strategies are known => opportunistic problem solving.
- Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
- ***All components have access to a shared data store, the blackboard.***
- Components may produce new data objects that are added to the blackboard.
- Components look for particular kinds of data on the blackboard, and may find these by pattern matching.

- Class
  - Blackboard
- Responsibility
  - Manages central data
- Collaborators
  - no

- Class
  - Knowledge Source
- Responsibility
  - Evaluates its own applicability
  - Computes a result
  - Updates Blackboard
- Collaborators
  - Blackboard

- Class
  - Control
- Responsibility
  - Monitors Blackboard
  - Schedules knowledge source activations
- Collaborators
  - Blackboard
  - Knowledge Source

General description

- Blackboard allows multiple processes (or agents) to communicate by reading and writing requests and information to a global data store.

- Each participant agent has expertise in its own field, and has a kind of problem solving knowledge (knowledge source) that is applicable to a part of the problem, i.e., the problem cannot be solved by an individual agent only.

- Agents communicate strictly through a common blackboard whose contents is visible to all agents.

- When a partial problem is to be solved, candidate agents that can possibly handle it are listed.

- A control unit is responsible for selecting among the candidate agents to assign the task to one of them.

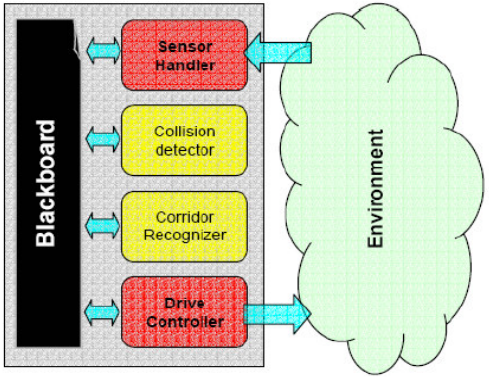
Example: speech recognition

- The main loop of Control started
  - Control calls nextSource() to select the next knowledge source
  - nextSource() looks at the blackboard and determines which knowledge sources to call
  - For example, nextSource() determine that Segmentation, Syllable Creation and Word Creation are candidate
  - nextsource() invokes the condition-part of each candidate knowledge source
  - The condition-parts of candidate knowledge source inspect the blackboard to determine if and how they can contribute to the current state of the solution
  - The Control chooses a knowledge source to invoke and a set of hypotheses to be worked on (according to the result of the condition parts and/or control data)
  - Apply the action-part of the knowledge source to the hypothesis
  - New contents are updated in the blackboard

Advantages & Disadvantages

- Advantages
  - Suitable when there are diverse sources of input data
  - Suitable for physically distributed environments
  - Suitable for scheduling and postponement of tasks and decisions
  - Suitable for team problem-solving approaches as it can be used to post problem subsomponents and partial results
- Disadvantages
  - Expensive
  - Difficult to determine partitioning of knowledge
  - Control unit can be very complex

Robot example



- An Experimental robot is equipped with four agents:
  - Sensor Handler Agent,
  - Collision Detector Agent,
  - Corridor Recognizer Agent and
  - Drive Controller Agent

(Includes the control software)

- Agents and blackboard form the control system. Agent cooperation is reached by means of the blackboard. Blackboard is used as a central repository for all shared information.
- Only two agents have an access to the environment: Sensor Handler Agent and Drive Controller Agent.
- There is no global controller for all of these agents, so each of them independently tries to make a contribution to the system during the course of navigation.
- Basically each of the four agents executes its tasks independently using information on the blackboard and posts any result back to the blackboard.

….patterns related to the communication infrastructure

- **Messaging**
  - Distribution Infrastructure
- **Publisher-Subscriber**
  - Distribution Infrastructure
- **Broker**
  - Distribution Infrastructure
- **Client Proxy**
  - Distribution Infrastructure
- **Reactor**
  - Event Demultiplexing and Dispatching
- **Proactor**
  - Event Demultiplexing and Dispatching

Messaging Pattern

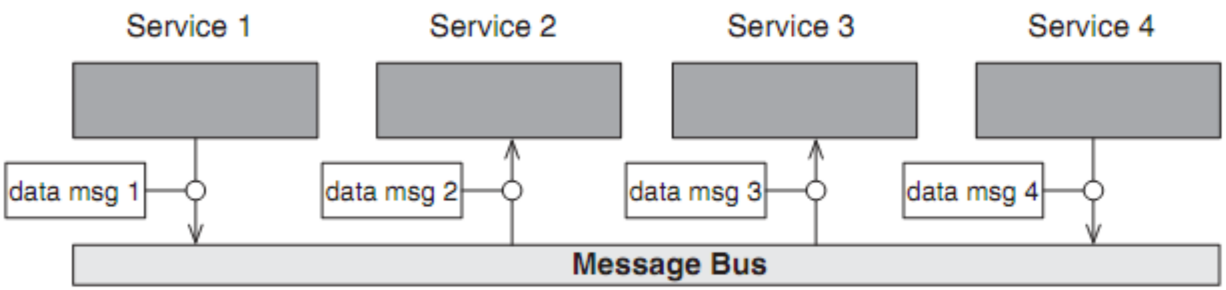
- Some distributed systems are composed of services that were developed independently.

- However, to form a coherent system, these services must interact reliably, but without incurring overly tight dependencies on one another.

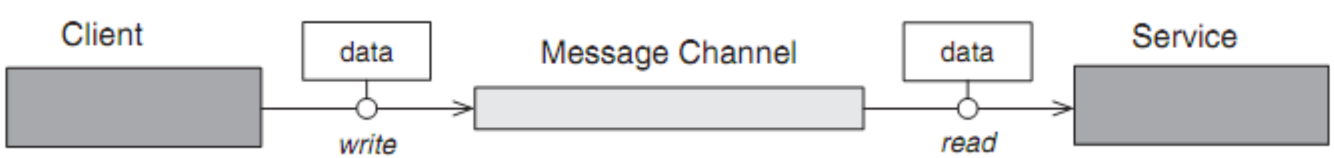
- **Solution:** *Connect the services via a message bus* that allows them to transfer data messages ***asynchronously***.
  - Encode the messages so that senders and receivers can communicate reliably without having to know all the data type information statically.

Messaging (continued)

- Message-based communication supports ***loose coupling*** between services in a distributed system.



- *Messages* only contain the data to be exchanged between a set of clients and services, so they do not know who is interested in them.
  - Therefore, another way is to connect the collaborating clients and services using a message channel that allows them to exchange messages, known as "Message Channel".



Publisher-Subscriber Pattern

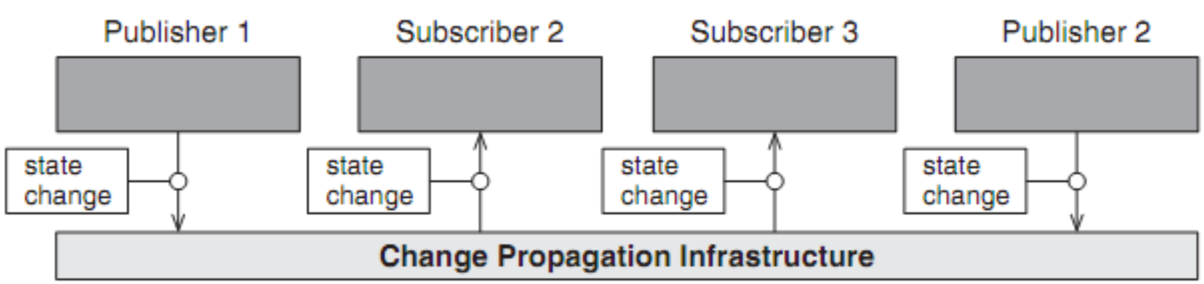
- Components in some distributed applications are loosely coupled and operate largely independently.

- if such applications need to propagate information to some or all of their components, a notification mechanism is needed to inform the components about state changes or other interesting events.

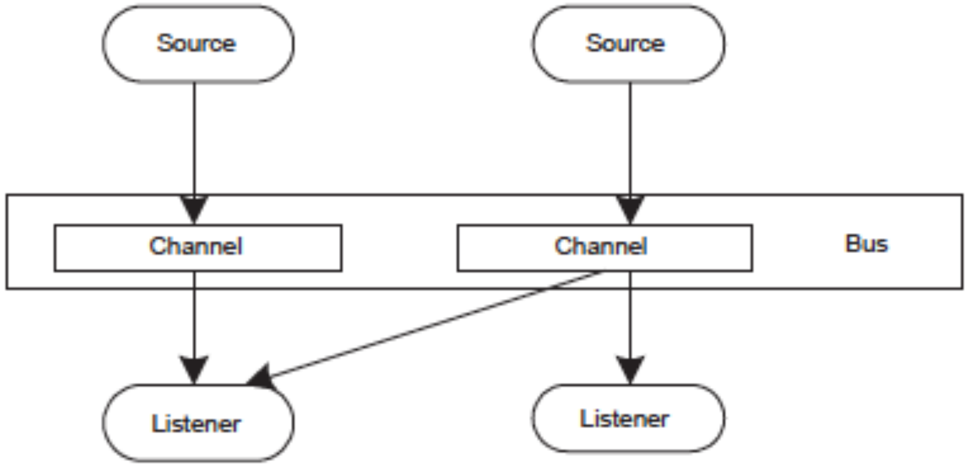
- **Solution:** Define a change propagation infrastructure that allows publishers in a distributed application to ***disseminate events*** that convey information that may be of interest to others.
  - Notify subscribers interested in those events whenever such information is published.

Publisher-Subscriber (continued)

- Publishers register with the change propagation infrastructure to inform it about what types of events they can publish.
- Similarly, subscribers register with the infrastructure to inform it about what types of events they want to receive.
- The infrastructure uses this registration information to route events from their publishers through the network to interested subscribers.



Event-Bus Pattern



- Event sources publish messages to particular channels on an event bus.
- Event listeners subscribe to particular channels .
- Listeners are notified of messages that are published to a channel to which they have subscribed.
- Generation and notification of messages is asynchronous: an event source just generates a message and may go on doing something else; it does not wait until all event listeners have received the message.

Broker Pattern

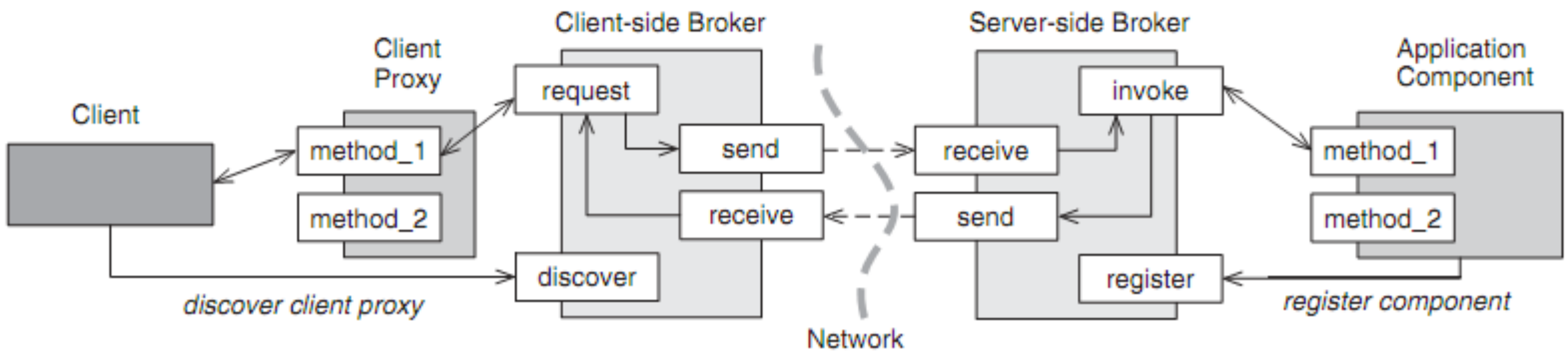
- Distributed systems face many challenges that do not arise in single-process systems.
  - However, application code should not need to address these challenges directly.

- Moreover, applications should be simplified by using a ***modular programming model*** that shields them from the details of networking and location.

- **<u>Solution</u>:** Use a federation of ***brokers to separate and encapsulate the details of the communication infrastructure*** in a distributed system from its application functionality.

- Define a ***component-based programming model*** so that clients can invoke methods on remote services as if they were local.

Broker (continued)



- In general a messaging infrastructure consists of two components:
  - A "Requestor" forwards request messages from a client to the local broker of the invoked remote component;
  - While an "Invoker" encapsulates the functionality for receiving request messages sent by a client-side broker and dispatching these requests to the addressed remote components.

Client-Proxy Pattern

- When constructing a client-side BROKER infrastructure for a remote component we must provide an abstraction that allows clients to access remote components using ***remote method invocation***.

- A “Client Proxy / Remote Proxy” represents a remote-component in the client’s address space.

- The proxy offers an identical interface that maps specific method invocations on the component onto the broker’s message-oriented communication functionality.

- ***Proxies allow clients to access remote component functionality as if they were collocated.***

Event Demultiplexing & Dispatching

- At its heart, distributed computing is all about ***handling and responding to events*** received from the network.

- There are patterns that describe different approaches for initiating, receiving, demultiplexing, dispatching, and processing events in distributed and networked systems.

- …two of these patterns: Reactor ; Proactor ;

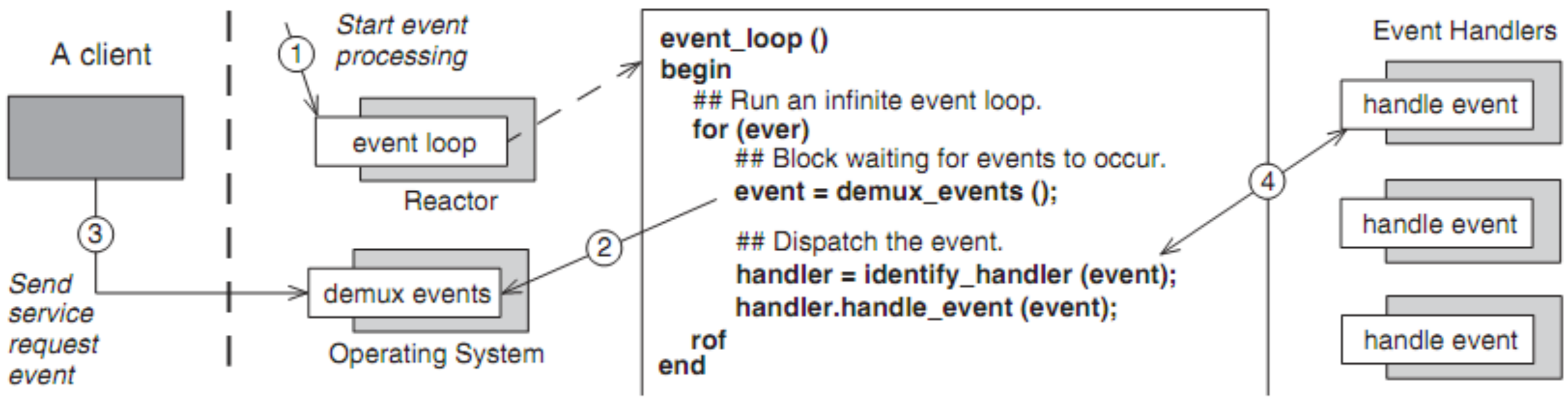
Reactor Pattern

- Event-driven software often
  - receives service request events from multiple event sources, which it demultiplexes and dispatches to event handlers that perform further service processing.

- Events can also arrive simultaneously at the event-driven application.
  - To simplify software development, events should be processed sequentially | synchronously.

Reactor (continued)

- **<u>Solution</u>:** *Provide an event handling infrastructure that waits on multiple event sources simultaneously for service request events to occur, but only demultiplexes and dispatches* ***one event at a time*** *to a corresponding event handler that performs the service.*



- It defines an event loop that uses an operating system event demultiplexer to wait synchronously for service request events.
- By ***delegating the demultiplexing of events to the operating system***, the reactor can wait for multiple event sources simultaneously without a need to multi-thread the application code.

Proactor Pattern

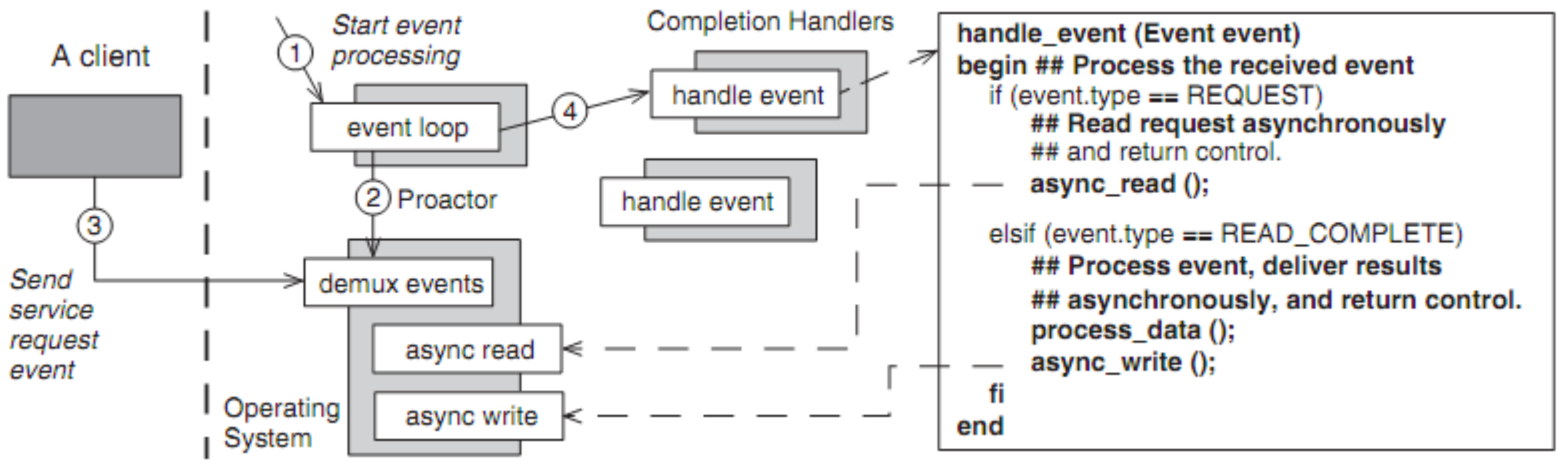
- To achieve the required performance and throughput, event-driven applications must often be able ***to process multiple events simultaneously***.

- However, resolving this problem via multi-threading, may be undesirable, due to the overhead of synchronization, context switching and data movement.

- **<u>Solution</u>:**
  - ***Split an application's functionality into***
    - ***asynchronous operations*** that perform activities on event sources and
    - **completion handlers** that ***use the results of asynchronous operations to implement application service logic***.
  - Let the operating system execute the asynchronous operations, but
  - execute *the completion handlers in the application's thread of control*.

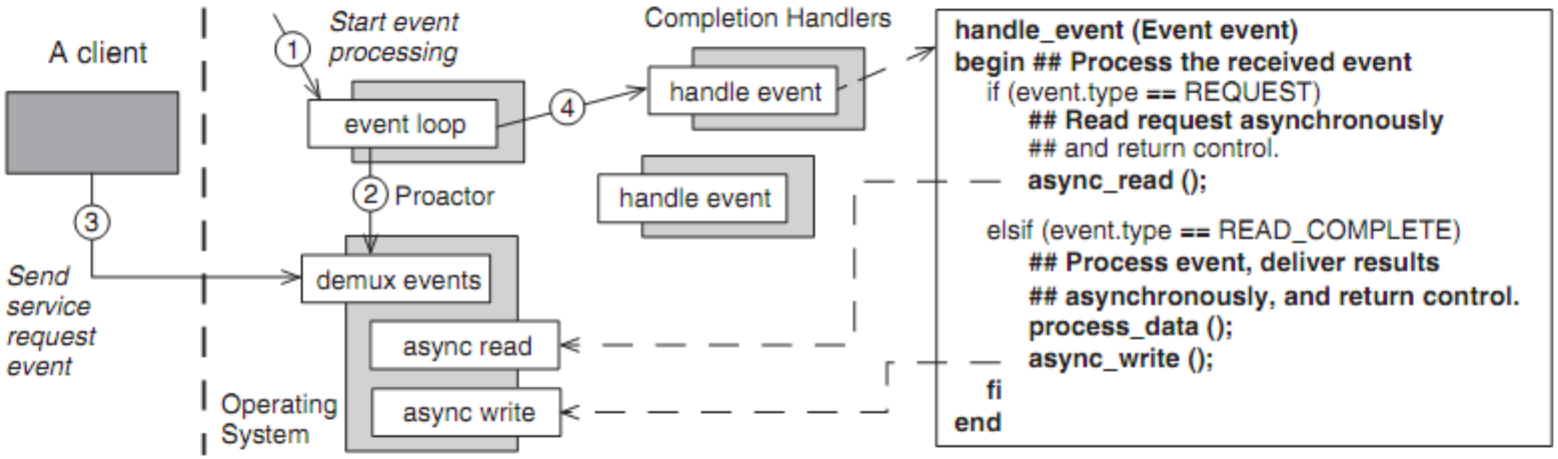
Proactor (continued)

- *A proactor component coordinates the collaboration between completion handlers and the operating system.*
  - It defines an event loop that uses an operating system event demultiplexer to wait synchronously for events that indicate the completion of asynchronous operations to occur.



- Initially **all completion handlers 'proactively' call an asynchronous operation to wait for service request events to arrive,** and then run the event loop on the proactor.

Proactor (continued)



- When such an event arrives, the proactor dispatches the result of the completed asynchronous operation to the corresponding completion handler.

- This handler then continues its execution, which may invoke another asynchronous operation.

Reactor vs. Proactor

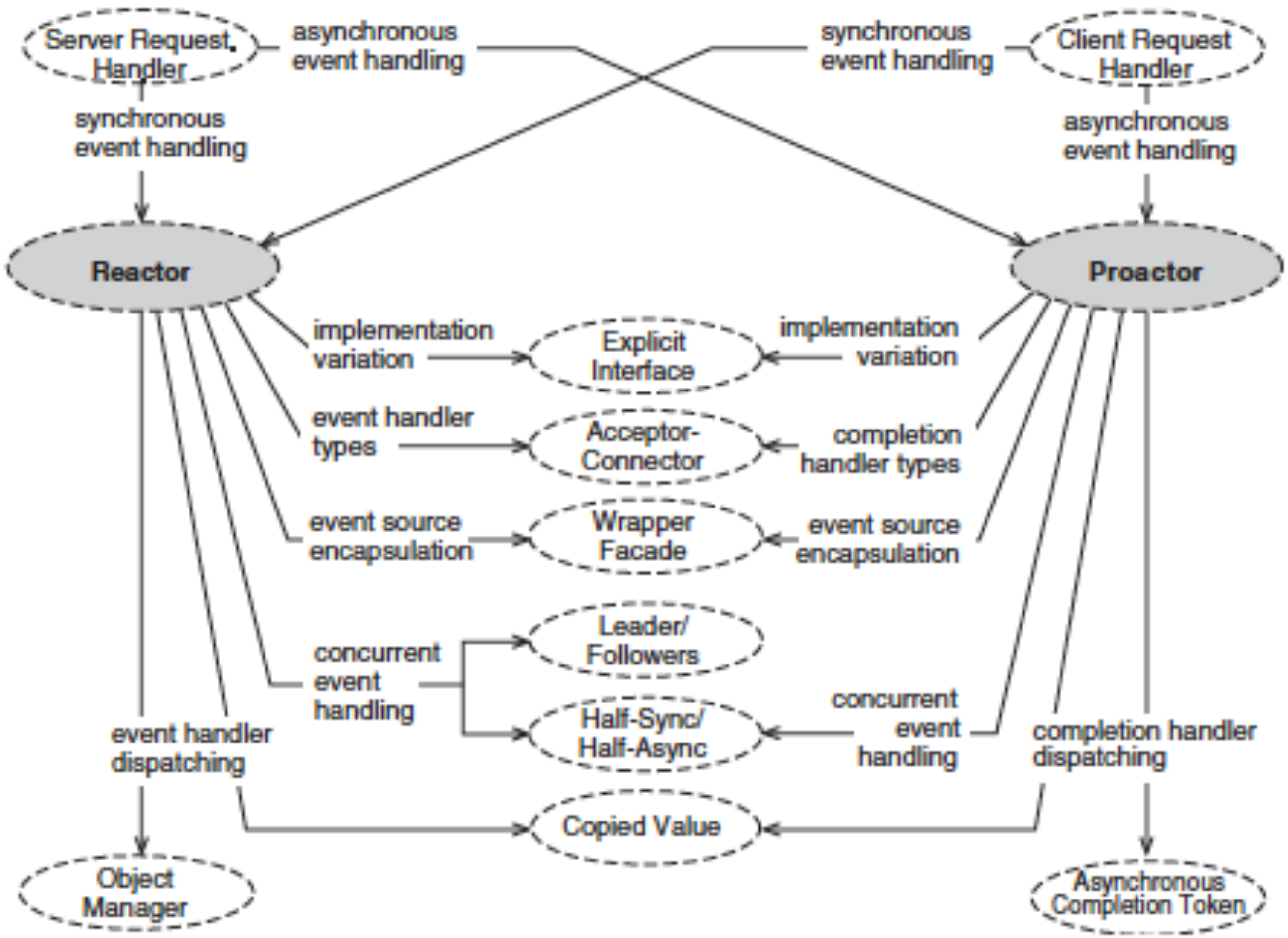
- Although both patterns resolve essentially the same problem in a similar context, and also use similar patterns to implement their solutions, the concrete event-handling infrastructures they suggest are distinct, due to the orthogonal forces to which each pattern is exposed.

- REACTOR focuses on simplifying the programming of event-driven software.
  - It implements a ***passive*** event demultiplexing and dispatching model in which services wait until request events arrive and then react by processing the events synchronously without interruption.
  - While this model scales well for services in which the duration of the response to a request is short, it can introduce performance penalties for long-duration services, since executing these services synchronously can unduly delay the servicing of other requests.

Reactor vs. Proactor (cont.)

- PROACTOR is designed to maximize event-driven software performance.
  - It implements a more *active* event demultiplexing and dispatching model in which services divide their processing into multiple self-contained parts and
  - ***proactively initiate asynchronous execution*** of these parts.
  - This design allows multiple services to execute concurrently, which can increase quality of service and throughput.

- REACTOR and PROACTOR are not really equally weighted alternatives, but rather are complementary patterns that trade-off programming simplicity and performance.
  - Relatively simple event-driven software can benefit from a REACTOR-based design, whereas
  - PROACTOR offers a more efficient and scalable event demultiplexing and dispatching model.


Server Request Handler
asynchronous event handling
synchronous event handling
Client Request Handler
synchronous event handling
asynchronous event handling
Reactor
Proactor
implementation variation
Explicit Interface
implementation variation
event handler types
Acceptor-Connector
completion handler types
event source encapsulation
Wrapper Facade
event source encapsulation
concurrent event handling
Leader/ Followers
Half-Sync/ Half-Async
concurrent event handling
event handler dispatching
completion handler dispatching
Copied Value
Object Manager
Asynchronous Completion Token

Middleware

- In computer science, a middleware is a software layer that resides between the application layer and the operating system.
- Its primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure, to coordinate how parts of applications are connected and how they inter-operate.
- Middleware also enables and simplifies the integration of components developed by different technology suppliers.

Middleware (continued)

- An Example of a middleware for distributed object-oriented enterprise systems is **CORBA**.

- Despite their detailed differences, middleware technologies typically follow one or more of three different communication styles:
  - Messaging
  - Publish/Subscribe
  - Remote Method Invocation

Referinte

- Frank Buschmann. Kevlin Henney.Douglas C. Schmidt.**Pattern-Oriented Software Architecture,**Volume 4: A Pattern Language forDistributed-Computing.Wiley & Sons, 2007

- George Coulouris. Jean Dollimore. Tim Kindberg.Gordon Blair. DISTRIBUTED SYSTEMS: Concepts and Design. Fifth Edition. Addison-Wesley.