

# Curs 13

Distributed Computing Patterns

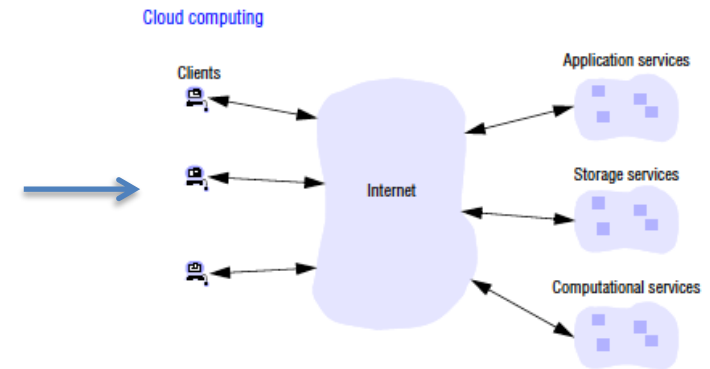
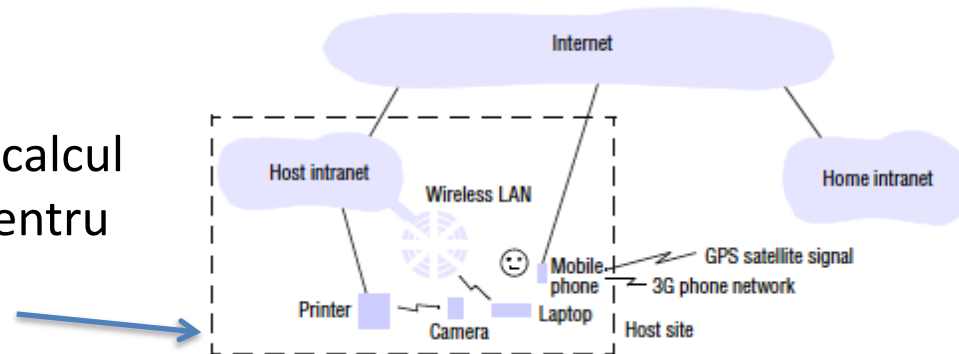
# Distributed systems

- Un sistem distribuit poate fi definit ca fiind format din **componente hardware si software localizate intr-o retea de calculatoare care comunica si isi coordoneaza actiunile doar bazat pe transmitere de mesaje**.
- Calcul distribuit (Distributed-Computing) = rezolvarea unor probleme folosind sisteme distribuite

# Tendinte

Influente semnificative:

- emergenta tehnologiilor de retea de scara larga
- emergenta necesitatilor crescute de calcul cuplata cu dorinta de asigura suport pentru mobilitatea utilizatorilor
- cresterea cererii de servicii multimedia
- perspectiva asupra sistemelor distribuite ca fiind o utilitate publica



# Caracteristici

## Concurenta:

- *se lucreaza cu programe care se executa concurent si care partajeaza resurse*

## No global clock:

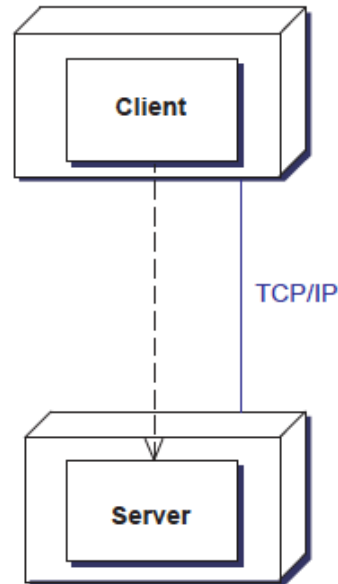
- *nu exista notiunea de timp global*
- aceasta este o consecinta directa a faptului ca singura modalitate de comunicare este transmiterea de mesaje printr-o retea

## Independent failures:

*Fiecare componenta a sistemului poate 'cadea' in mod independent lasand celelalte in starea de executie ('run').*

- toate retelele de calculatoare pot esua('fail') si este responsabilitatea proiectantilor sistemului sa gestioneze efectele in aceste cazuri si sa asigure masuri de reorganizare.
- Aceste tipuri de probleme conduc la izolarea computerelor conectate prin aceste retele aflate in starea de 'fault' – dar nu inseamna si oprirea celorlalte computere;
- Programele pot sa continue sa functioneze dar nu pot detecta daca retea a cazut sau este doar incetinita.
- Similar, 'caderea' unui computer, sau oprirea neasteptata a unui software undeva in sistem (*a crash*), nu este imediat facuta cunoscuta celorlalte componente cu care acesta comunica

# Client-Server pattern



- O componenta de tip server care furnizeaza servicii catre mai multe componente client.
- O componenta client cere servicii de la componenta server.
- Serverele sunt active permanent 'ascultand' cererile de la clienti.

# Starea(State) in sablonul Client-server

Clientii si serverele lucreaza in general in sesiuni ('sessions').

–**stateless server** -- starea unei sesiuni (**session state**) este gestionata de catre client. Aceasta stare (client) este trimisa impreuna cu fiecare cerere. In aplicatiile web, *session state* poate fi stocata ca si parametrii URL, in campuri ascunse sau folosind cookies (obligatoriu pentru arhitecturile REST folosite pentru aplicatii web).

–**stateful server** -- starea unei sesiuni (**session state**) este mentinuta la nivel de server si este asociata cu ID clientului

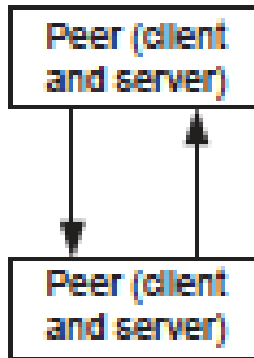
Modalitatea de gestionare a starii unei sesiuni influenteaza tranzactiile, scalabilitatea si gestiunea erorilor (*fault handling*).

- Tranzactiile trebuie sa fie
  - atomice si sa asigure consistenta starii,
  - izolate (sa nu afecteze alte tranzactii)
  - durabile
- *fault handling* => starea mentinuta la nivelul clientului implica faptul ca toata informatia se va pierde in cazul in care clientul esueaza (*fails*).
- Securitatea poate fi afectata daca starea se mentine la nivel de client pentru ca informatia se transmite de fiecare data (la fiecare *request*).
- Scalabilitatea poate fi reduisa daca starea se mentine la nivelul serverului '*in-memory*' – multi client, multe cereri => necesar de memorie crescute.

# Peer-to-peer pattern

- Poate fi privit ca si un sablon Client-Server **simetric**:
  - un nod (*peer*) poate functiona ca si un client – care cere servicii de la alte componente sau ca si server care furnizeaza servicii pentru altii.
  - rolul unui nod se poate schimba in mod dinamic
- Atat clientii cat si servere folosesc in mod uzual multithreading.
- Serviciile pot fi implicite (de exemplu prin intermediul unui stream de conectare) in locul unei cerere (request) trimise prin invocare directa.
- Un *peer* care actioneaza ca si server isi pot informa colegii (peers) care activeaza ca si client de aparitia unor evenimente; clientii pot fi informatii folosind de exemplu o magistrala de evenimente (event-bus).

# Example



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



# Caracteristici

- **Performanta** creste atunci cand numarul de noduri creste dar scade atunci cand sunt prea putine

## Avantaje

- Nodurile pot folosi capacitatea intregului sistem chiar daca fiecare are o capacitate proprie limitata.
  - este un **cost individual mic cu beneficiu mare** obtinut prin partajare
- Overhead-ul de administrare este scazut pentru ca retelele peer-to-peer se organizeaza intern (self-organizing)
- Asigura **scalabilitate** foarte buna si este rezilienta la esec/caderea(failure) componentelor individuale.
- Configurarea sistemului se poate schimba **dinamic**: un *peer* poate intra sau pleca in timp ce sistemul functioneaza.

## Dezavantaje

- nu exista garantia calitatii serviciilor ( *no guarantee about quality of service*) deoarece nodurile coopereaza voluntar
- nu exista garantia securitatii ( *security is difficult to guarantee*) deoarece nodurile coopereaza voluntar

## Forwarder-Receiver

- **Communication Pattern**
  - Forwarder-Receiver furnizeaza in mod transparent comunicarea inter-proces pentru sistemele sistem cu model de interactiune de tip peer-to-peer.
  - Foloseste *forwarders* si *receivers* pentru a decupla nodurile de mecanismul de comunicare de baza (the underlying mechanism).

# Architectural patterns

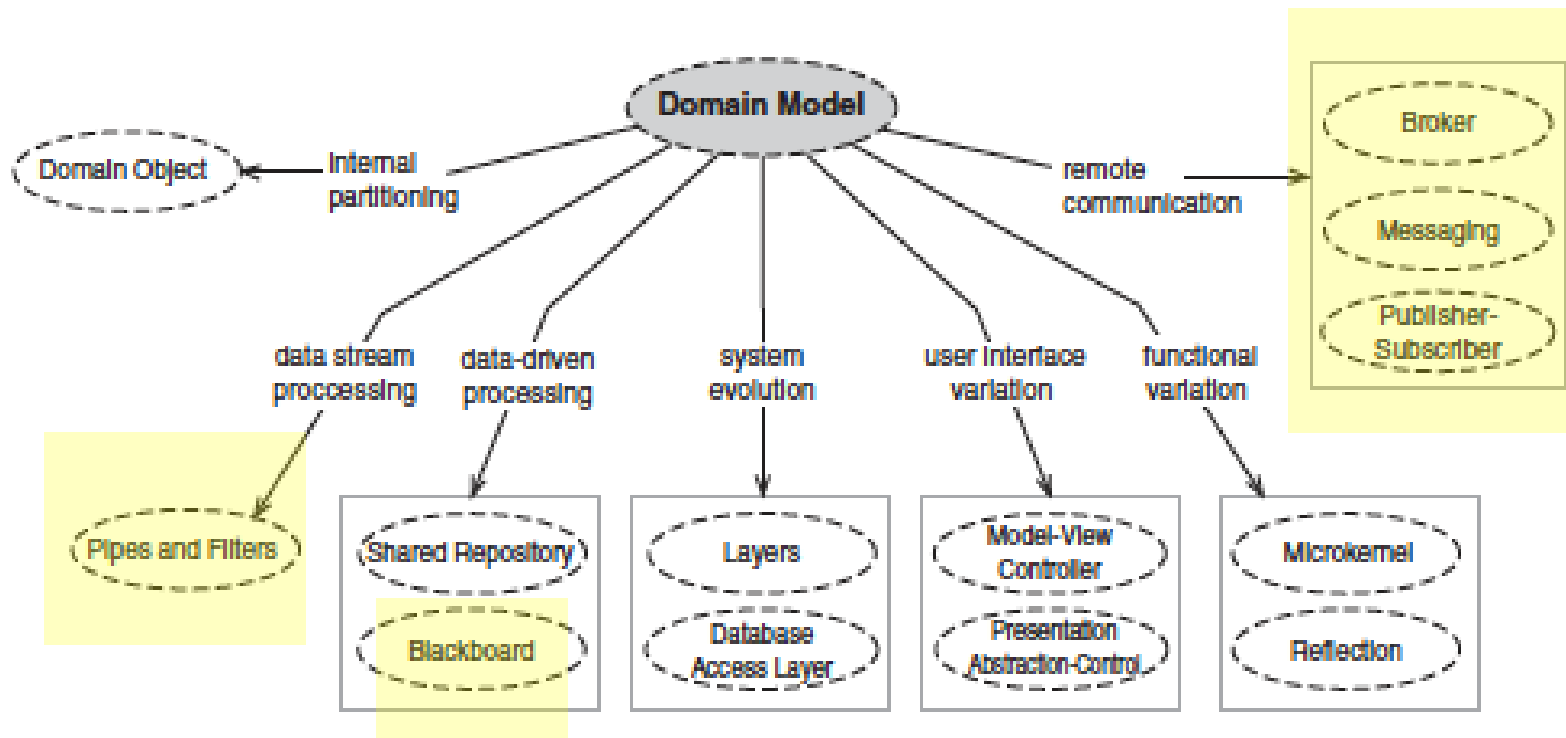
Frank Buschmann. Kevlin Henney. Douglas C. Schmidt. *Pattern-Oriented Software Architecture*, Volume 4:

A Pattern Language for Distributed-Computing. Wiley & Sons, 2007

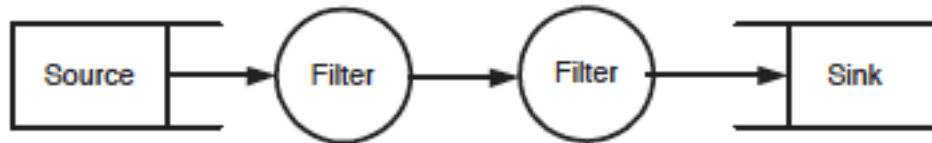
Un sablon arhitectural este un concept care  
rezolva si delimiteaza anumite elemente esentiale de coeziune ale unei arhitecturi  
software.

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

# Conexiunea cu Domain Layer

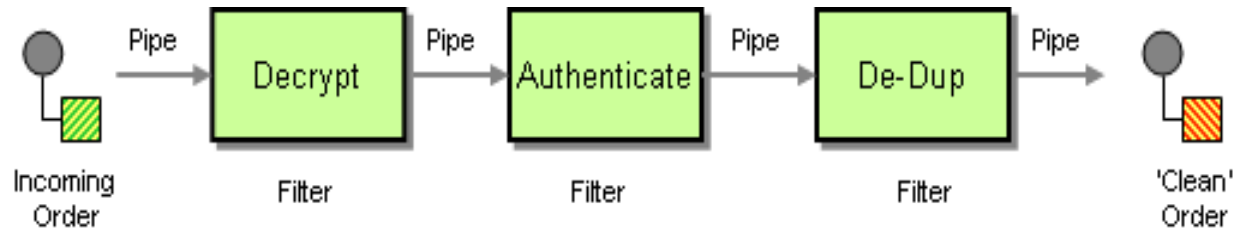


# Pipe-Filter Pattern



- furnizeaza pentru un sistem o structura care produce un ***stream de date***
- fiecare pas de procesare este incapsulat intr-o componenta de tip *filter*
- Datele sunt transferate prin pipes
  - *pipe* – leaga 2 filtre
  - *pipes* pot fi utilizate pentru sincronizare sau pentru buffering

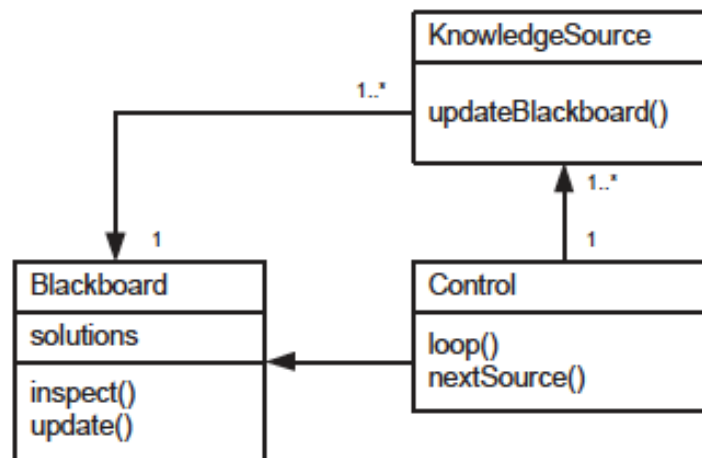
# Exemplu



- se poate utiliza pentru a divide taskuri de procesare mari intr-o secventa de componente independente mai mici de procesare (Filters) care sunt conectate prin canale (Pipes).
- fiecare filtru expune o interfata simpla – primeste mesaje de la *inbound pipe*, proceseaza mesajul si publica rezultatul la outbound pipe.
- pentru ca toate componentele utilizeaza aceeasi interfata externa ele pot fi compuse in diferite solutii prin conectarea componentelor la pipe-uri diferite
- se pot adauga filtre sau rearanja in secvente noi fara a se schimba filtrele (in interior).

# Blackboard Pattern

arata cum se poate rezolva o problema complexa precum recunoasterea de imagini sau de limbaj prin impartirea in subsisteme mai mici specializate care pot rezolva problema impreuna.



- mai multe subsisteme specializate asambleaza cunostinte pentru a construi partial sau aproximativ solutia
- ***Toate componentele au acces la ansamblul de date partajate = the blackboard.***
- Componentele pot produce date noi care sunt adaugate pe blackboard.
- Componentele cauta anumite date pe blackboard, folosind e.g. pattern matching.

# Componente

- 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



# Descriere generala

- Blackboard permite mai multor procese (agenti) sa comunice prin citirea si scrierea de cereri si informatii catre un depozit de date globale.
- **Fiecare agent participant are o expertiza in propriul domeniu**, si are un anumit tip de cunoastere referitoare la rezolvarea problemei (knowledge source) care se poate aplica la o parte a problemei, i.e., problema nu se poate rezolva de catre un singur agent.
- **Agentii comunica strict prin intermediul unei *common blackboard*** care este vizibil tuturor agentilor.
- Atunci cand o problema partiala trebuie rezolvata potentialii agenti care ar putea sa o rezolve sunt listati.
- O **unitate de control** este responsabila pentru selectarea agentilor si atribuirea de sarcini acestora.

# Exemplu: speech recognition

## Ciclul principal de rezolvare

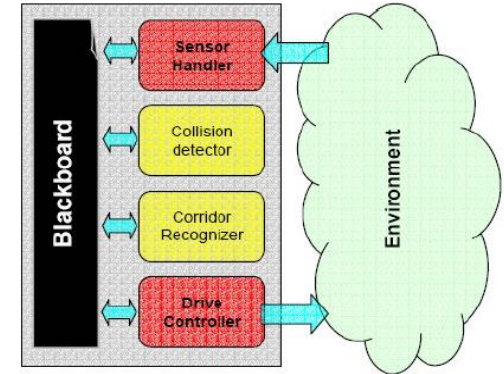
- **Control** calls **nextSource()** to select the next knowledge source
- **nextSource()** looks at the blackboard and determines which knowledge sources to call
- For example, **nextSource()** determine that **Segmentation**, **Syllable Creation** and **Word Creation** are candidate
- **nextsource()** 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***

# 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.

# Avantaje & Dezavantaje

- Avantaje
  - potrivit pentru diverse surse de date/procesare
  - potrivit pentru medii distribuite
  - **potrivit pentru planificarea taskurilor si a deciziilor**
  - **potrivit pentru abordari de rezolvare in echipa** – se posteaza subcomponente si rezultate partiale
  - util pentru probleme pentru care nu sunt cunoscute strategii de rezolvare deterministe => **opportunistic problem solving**.
- Dezavantaje
  - Scump
  - Dificil de a determina partitionarea pentru knowledge
  - Control unit poate fi foarte complexa

# ....patterns legate de infrastructura de comunicare

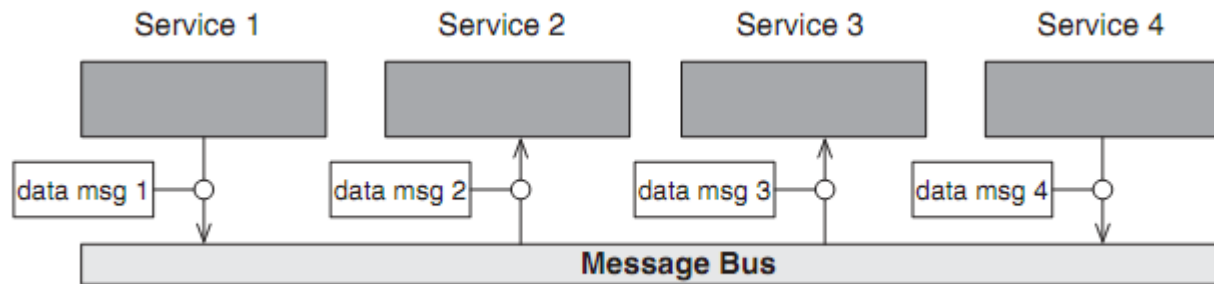
- **Messaging**
  - Distribution Infrastructure
- **Publisher-Subscriber**
  - Distribution Infrastructure
- **Broker**
  - Distribution Infrastructure
- **Client Proxy**
  - Distribution Infrastructure
- **Reactor**
  - Event Demultiplexing and Dispatching
- **Proactor**
  - Event Demultiplexing and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

# Messaging Pattern

- Unele sisteme distribuite sunt compuse din servicii care sunt dezvoltate independent.
- Pentru a forma un sistem coerent aceste servicii trebuie sa interactioneze intr-un mod fiabil fara a implica totusi o dependenta stransa intre ele.
- **Solutie:** Conectarea printr-un canal de mesaje (*a message bus*) care sa permita transferul de date ***asincron***.
  - mesajele se codifica astfel incat comunicarea sa se poata face fara sa fie nevoie de sa fie cunoscute toate informatiile legate de tipurile de date.

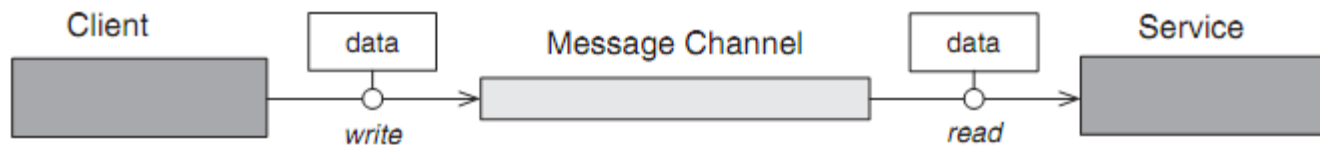
# Messaging

- Message-based communication suporta *loose coupling*



- Mesajele* contin doar datele care pot fi interschimbate intre setul de clienti si servicii – nu si cine este interesat de acestea.

=> conectarea clientilor si serviciilor printr-un canal care permite schimbul de mesaje “Message Channel”.



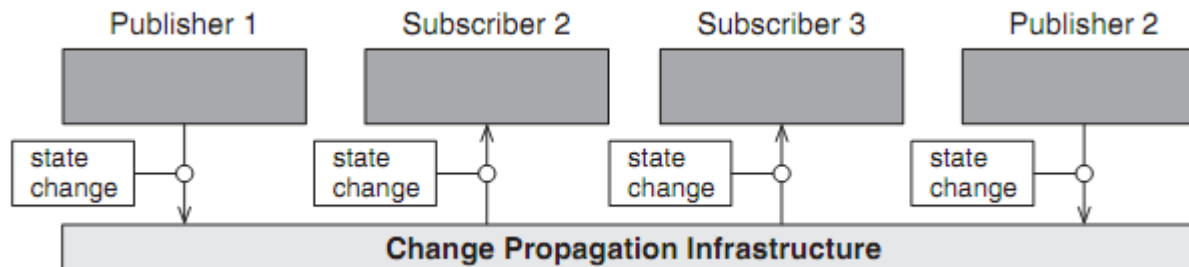
# Publisher-Subscriber Pattern

- Componentele din anumite aplicatii distribuite sunt cuplate slab si opereaza in general independent.
- pentru a propaga informatii in asemenea aplicatii se poate folosi un mechanism de notificare prin care sa se faca informari referitoare la modificari de stare sau referitoare la aparitia unor evenimente.
- **Solutie:** Definirea unei infrastructuri de propagare a informatiei care permite editorilor sa disemineze evenimente care contin informatie care ar putea sa intereseze alti actori.
  - se notifica abonatii inregistrati pentru a primi anumite tipuri de notificari

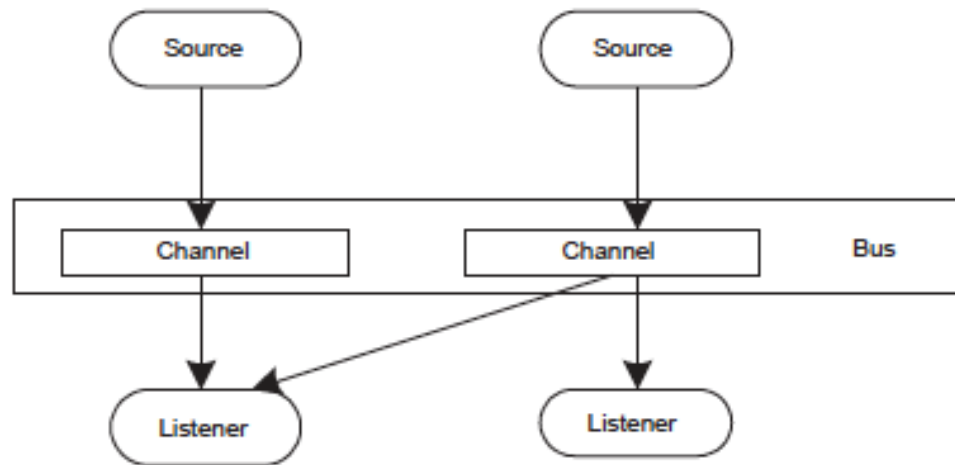


# Publisher-Subscriber

- Editorii inregistreaza ce tipuri de evenimente publica.
- Abonatii inregistreaza de ce tipuri de evenimente sunt interesati.
- Infrastructura foloseste informatiile inregistrate pentru a transmite prin retea evenimentele de la editori la abonatii inregistrati a fi interesati.



# Event-Bus Pattern

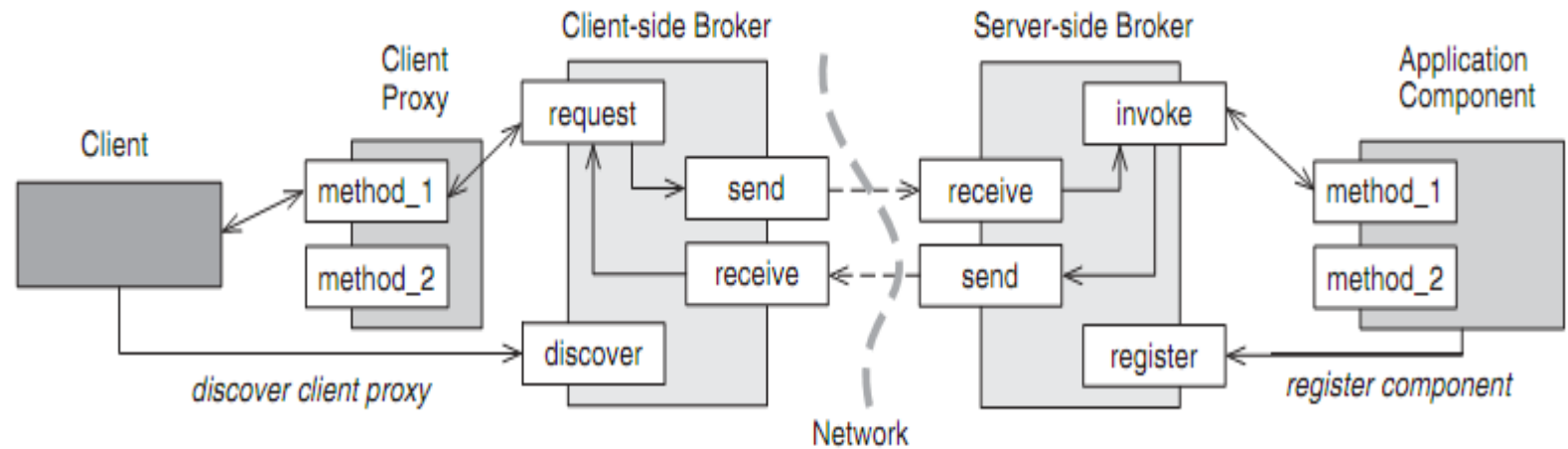


- Sursele de evenimente (*Event sources*) publica mesaje pe canale particulare pe un (*event bus*)
- Event listeners se aboneaza la anumite canale.
- Listenerii sunt notificati referitor la canalele care sunt publicate pe un canal la care s-au abonat.
- Generarea si notificarea de mesaje este **asincrona** :
  - un event source genereaza un mesaj – nu asteapta ca acesta sa fie primit de catre listeneri

# Broker Pattern

- Sistemele distribuite pot avea dificultati (provocari) care nu apar in sistemele cu un proces
  - Important – codul aplicatiile nu trebuie sa trateze aceste probleme in mod direct => intermediari (brokers)
- aplicatiile trebuie simplificate prin modularizare *modular programming model* care poate ascunde detaliile de retea si locatie
- **Solutie:** Folosirea unei federatii de brokeri
  - *brokers to separate and encapsulate the details of the communication infrastructure* in a distributed system *from its application functionality*.
- Definirea unui model de tip *component-based programming model prin care clientii sa poate sa invoce metode sau servicii remote ca si cum ar fi locale*

# Broker



- .

# Client-Proxy Pattern

- Atunci cand se construiesc o infrastructura de tip broker *client-side* pentru o componenta *remote* trebuie sa se asigure o abstractizare care permite clientilor sa acceseze acele componente folosind ***remote method invocation***.
- un “Client Proxy / Remote Proxy” reprezinta o componenta remote- in the spatial de adrese al clientului.
- Proxy ofera o interfata identica care mapeaza invocarile de metode specifice catre functionalitatea orientate catre transmiterea de mesaje ale brokerului.
- ***Proxies allow clients to access remote component functionality as if they were collocated.***

# Event Demultiplexing & Dispatching

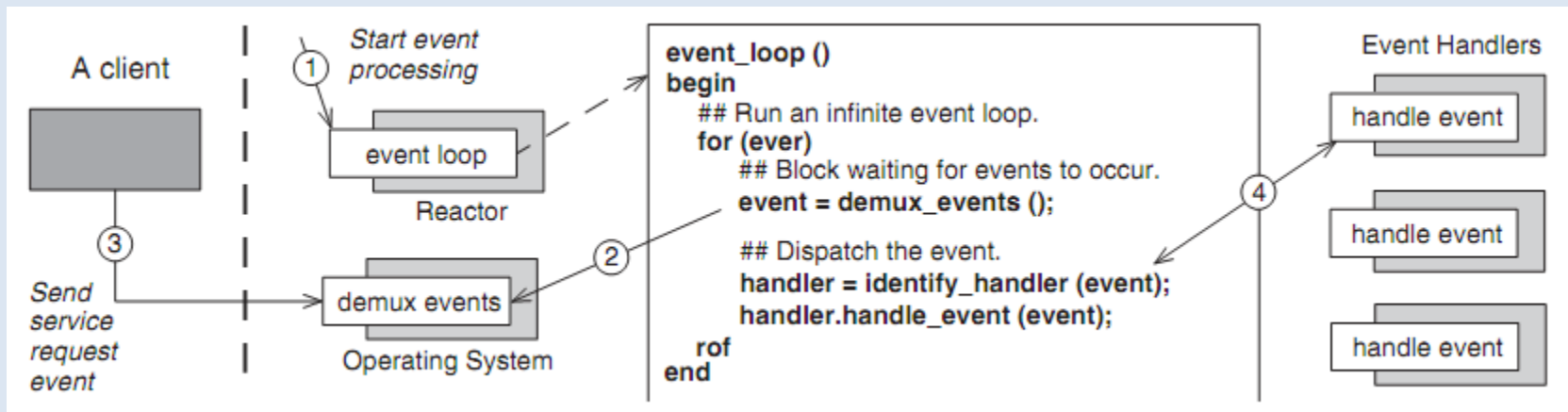
- At its heart, distributed computing is all about handling and responding to events received from the network.
- There are patterns that describe different approaches for initiating, receiving, demultiplexing, dispatching, and processing events in distributed and networked systems.
- ...two of these patterns:
  - Reactor ... synchronous
  - Proactor ... asynchronous

# 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

- **Solution:** *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.

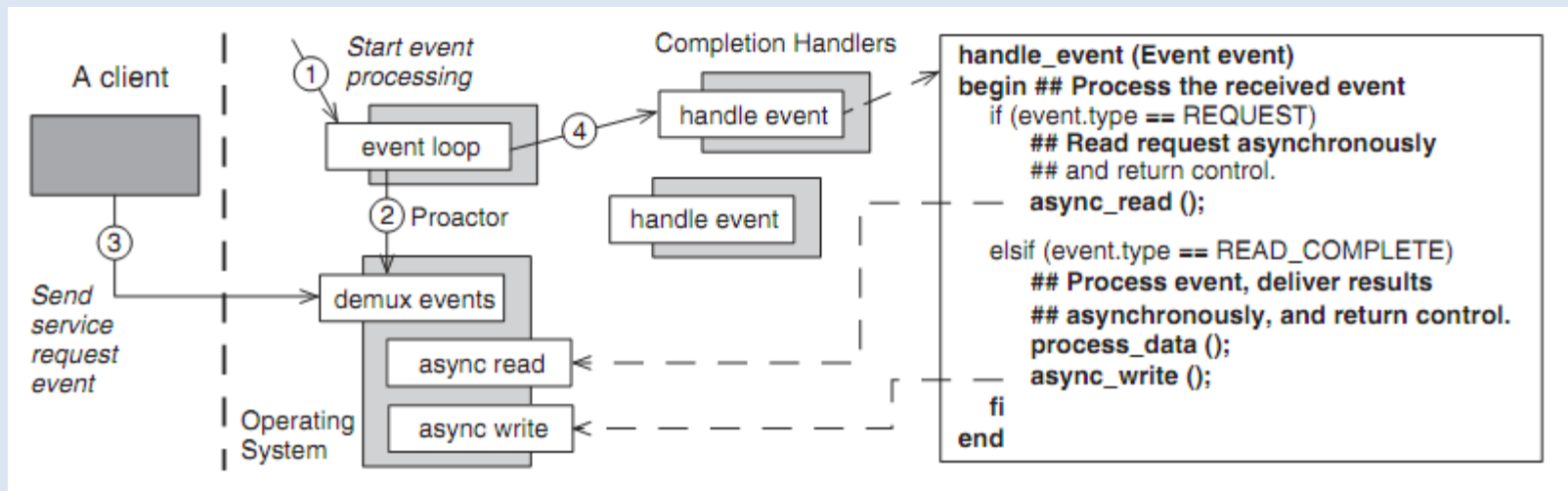


# Proactor Pattern

- 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.**
- **Solution:**
  - *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.*

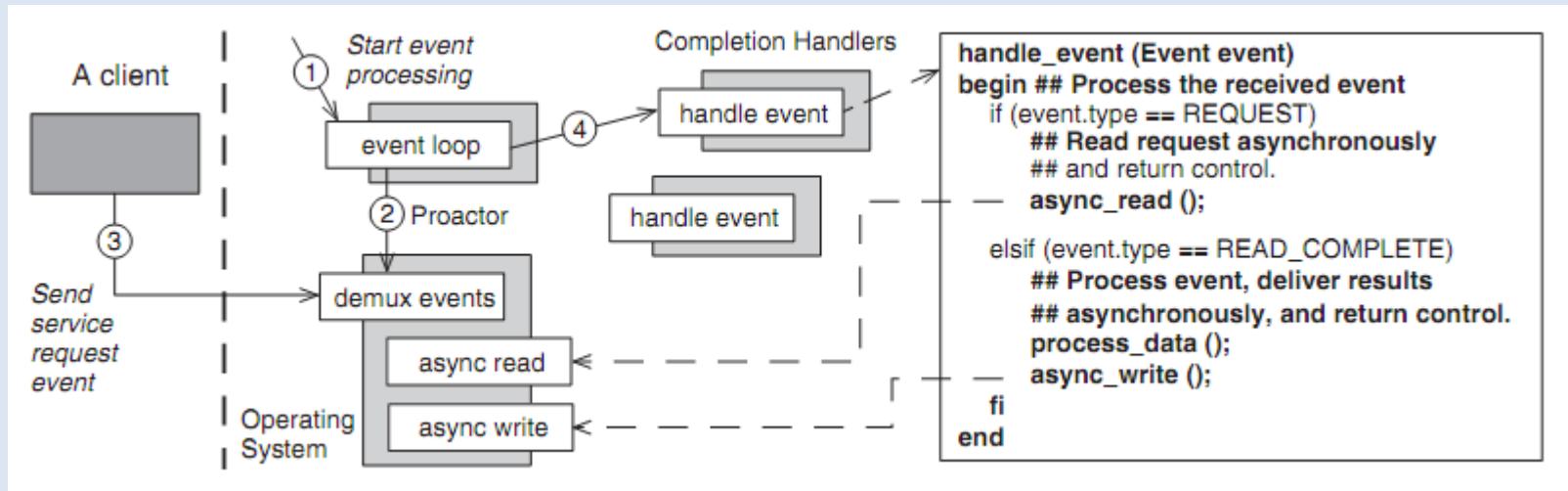
# Proactor

- 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



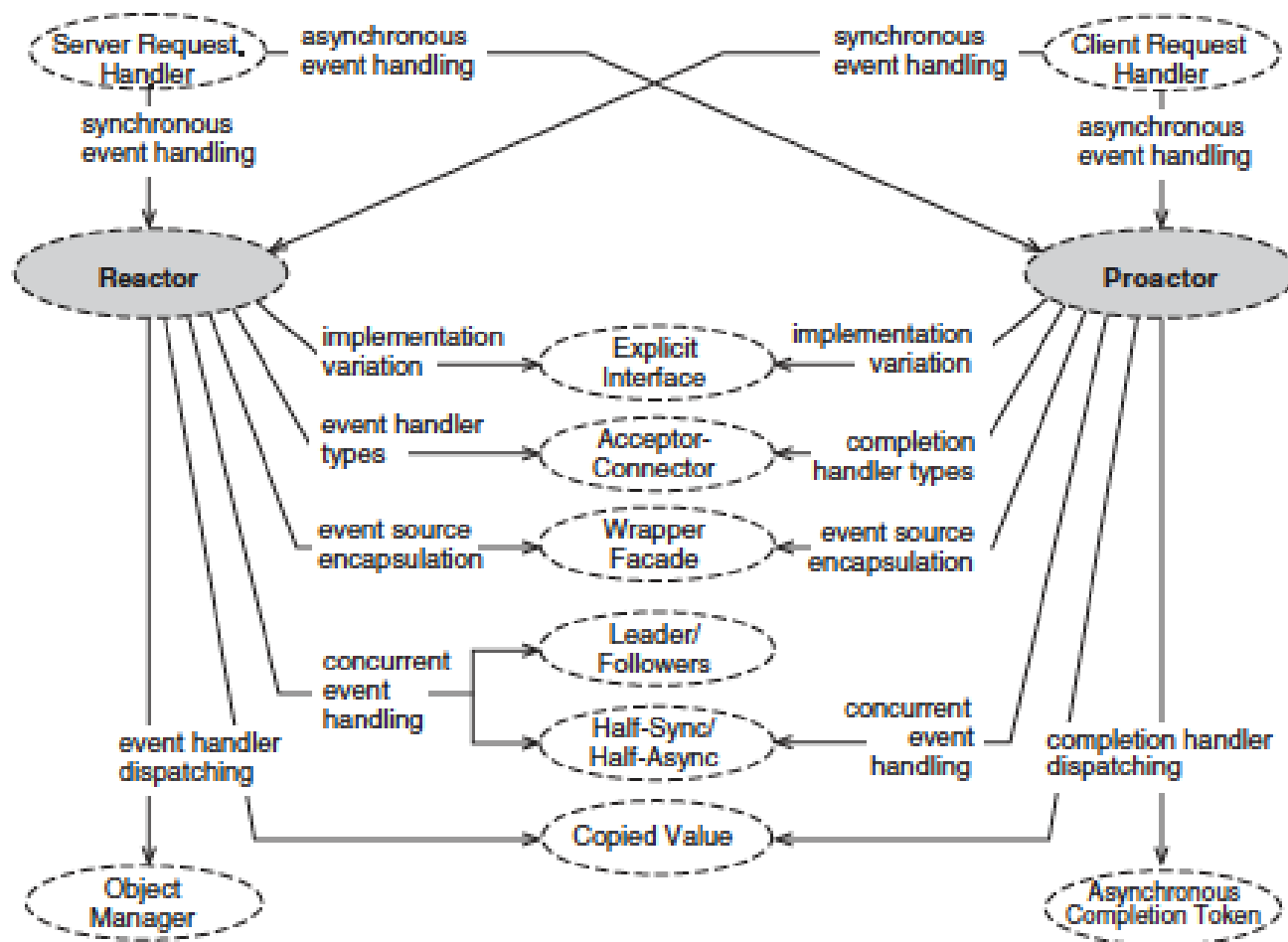
- When such an event arrives, the proactor dispatches the result of the completed asynchronous operation to the corresponding completion handler.
- This handler then continues its execution, which may invoke another asynchronous operation.

# 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

- 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.



# 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

Examples of a middleware for distributed object-oriented enterprise systems: CORBA, SOAP.

- 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 for Distributed-Computing. Wiley & Sons, 2007
- George Coulouris. Jean Dollimore. Tim Kindberg. Gordon Blair. **DISTRIBUTED SYSTEMS: Concepts and Design**. Fifth Edition. Addison-Wesley.