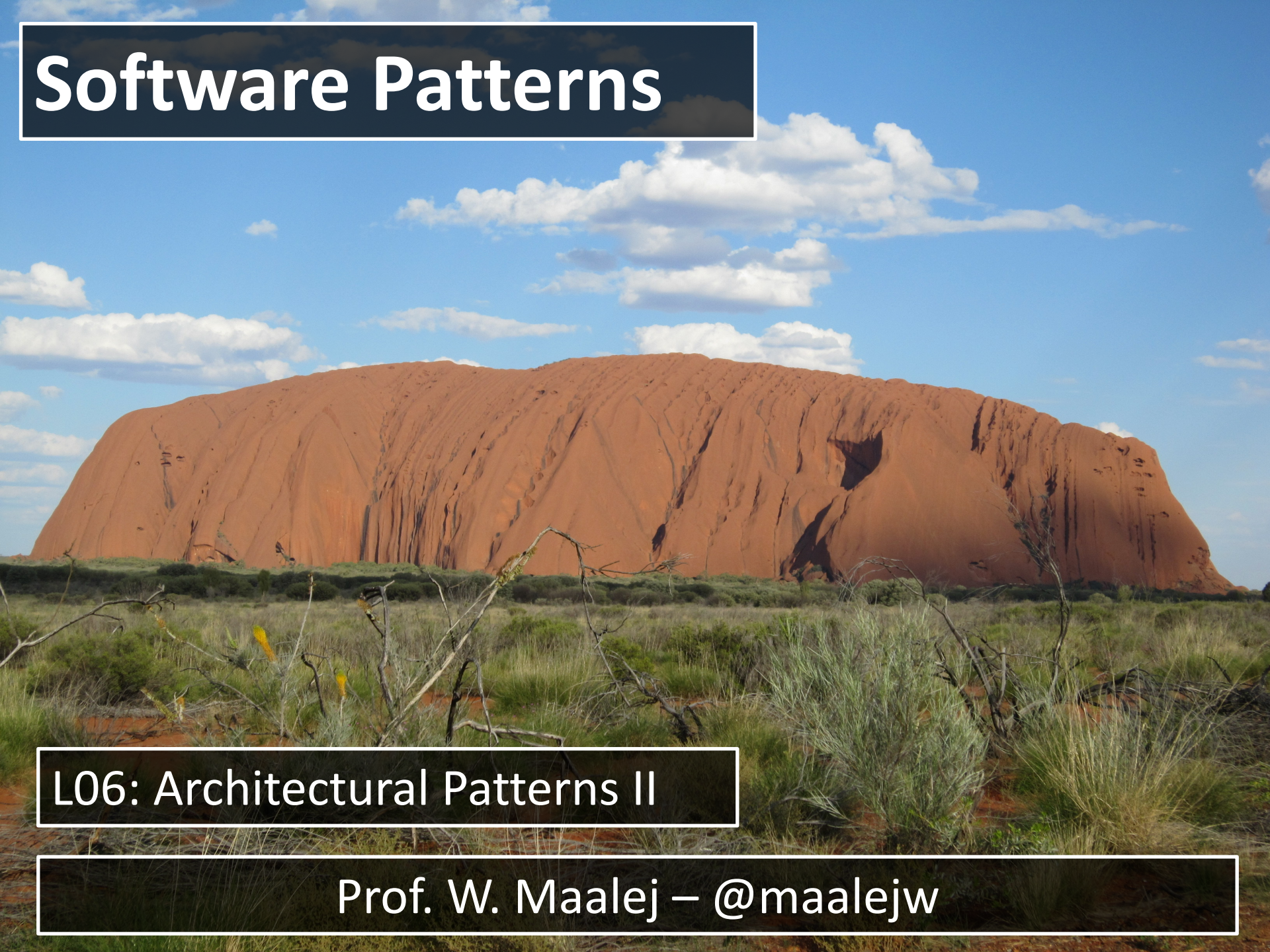
Software Patterns



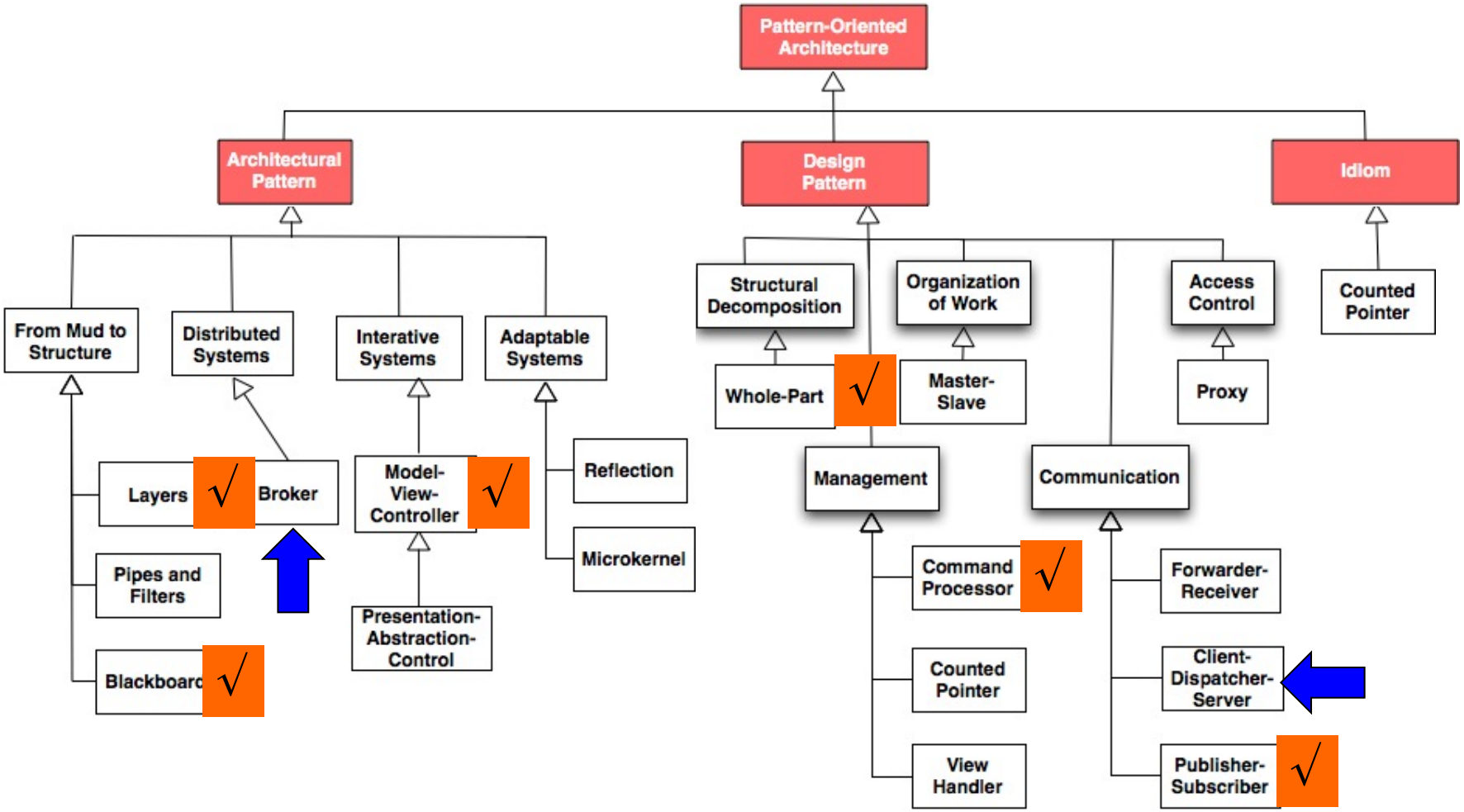
L06: Architectural Patterns II

Prof. W. Maalej – @maalejw

Outline of the talk

1. **Motivation**
2. Client-Dispatcher-Server Pattern
3. Broker Pattern

Posa-taxonomy (gang-of-five) (pattern-oriented software architecture)



Patterns for Distributed Systems

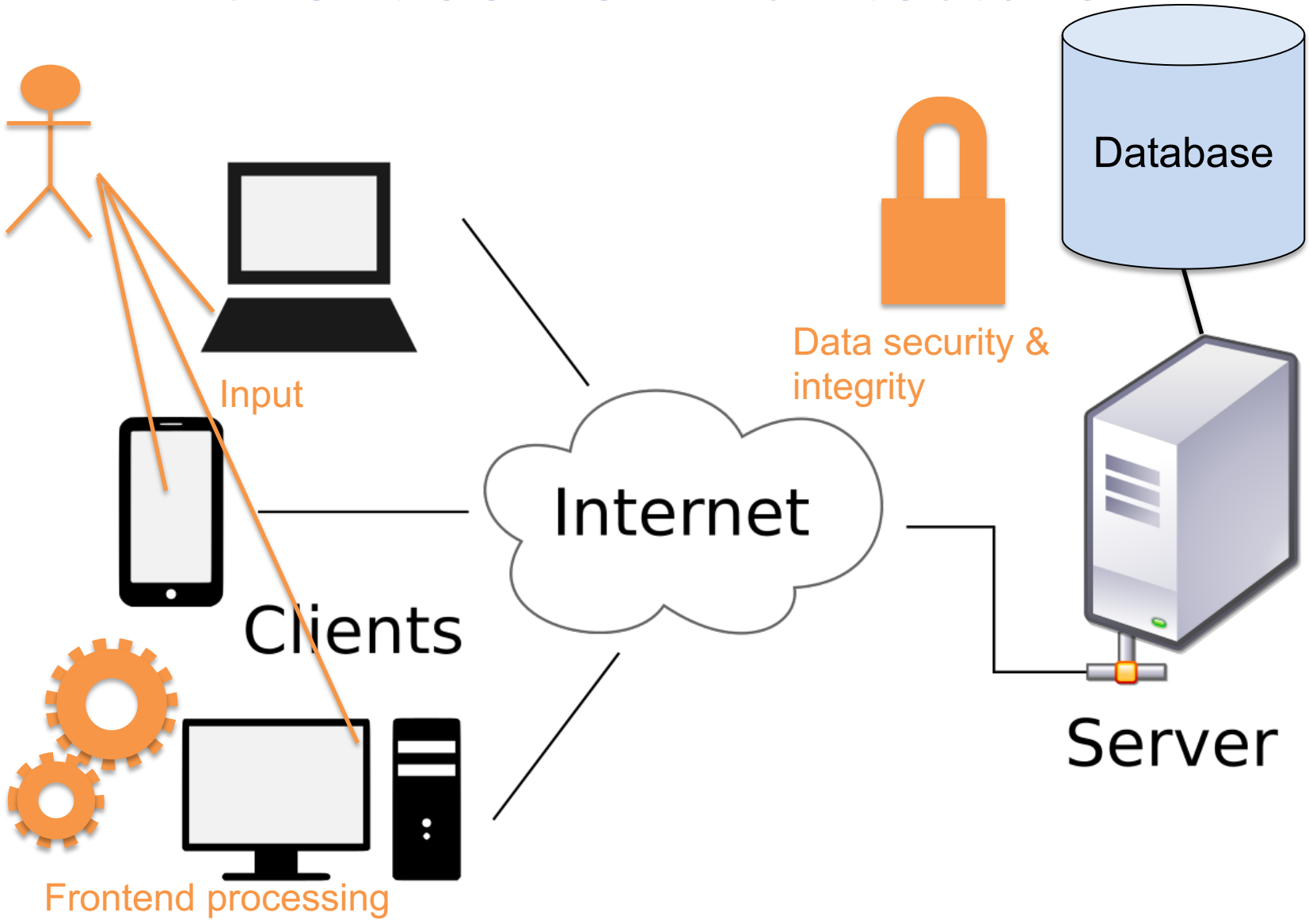
Mini-intro: distributed systems

- Trend: From Monolithic to Distributed Systems
  - **Multiprocessors** (Shared memory systems, MISD): Processors with multiple CPUs, graphic cards with 128 cores (providing more than 300 GFLOPs), multi-processors with more than 8 cores on a chip
  - **Networks** (Message-based Systems, MIMD): Heterogeneous systems connected via a network (Internet, LAN network, home)
- Some applications are distributed by their definition:
  - Examples: Client-server systems, Web applications, Database applications

Advantages of distributed systems

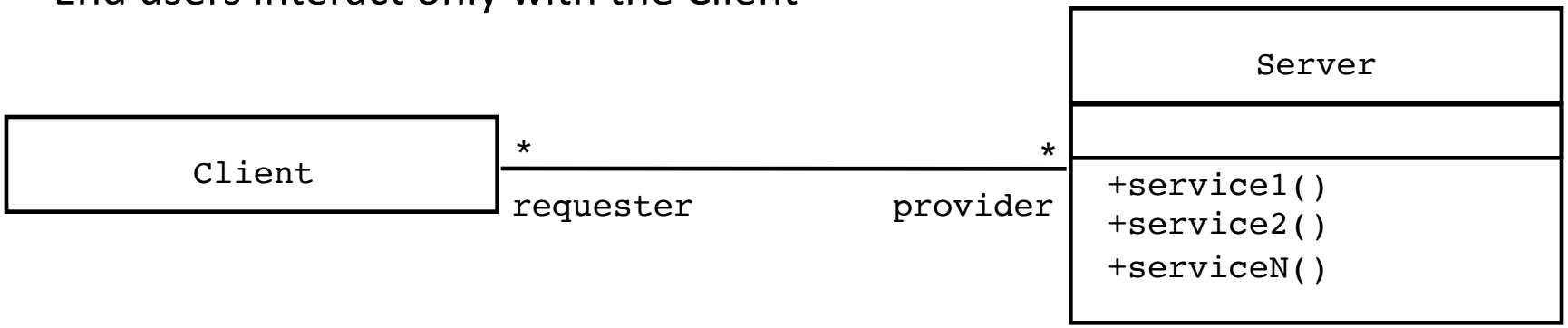
- **Economics**: Computer networks offer a better price/performance ratio than a single large computer (mainframe)
- **Scalability**: Multiprocessors and networks are easily scalable
- **Better Performance**: Distributed applications can increase their performance by using resources all across the network
- **Higher Reliability**: A machine on a network can crash without bringing down the rest of the application

Client Server Architecture



Client/Server architectural styles

- The Client/Server architectural style is a special case of the layered architecture style:
- One or more **Servers provide** services to instances of subsystems, called **Clients**
- Each **Client calls** a service offered by the Server; the Server performs the service and returns the result to the Client
- The Client knows the **interface** of the Server
- The Server **does not know** the interface of the Client
- The response in general is immediate
- End users interact only with the Client

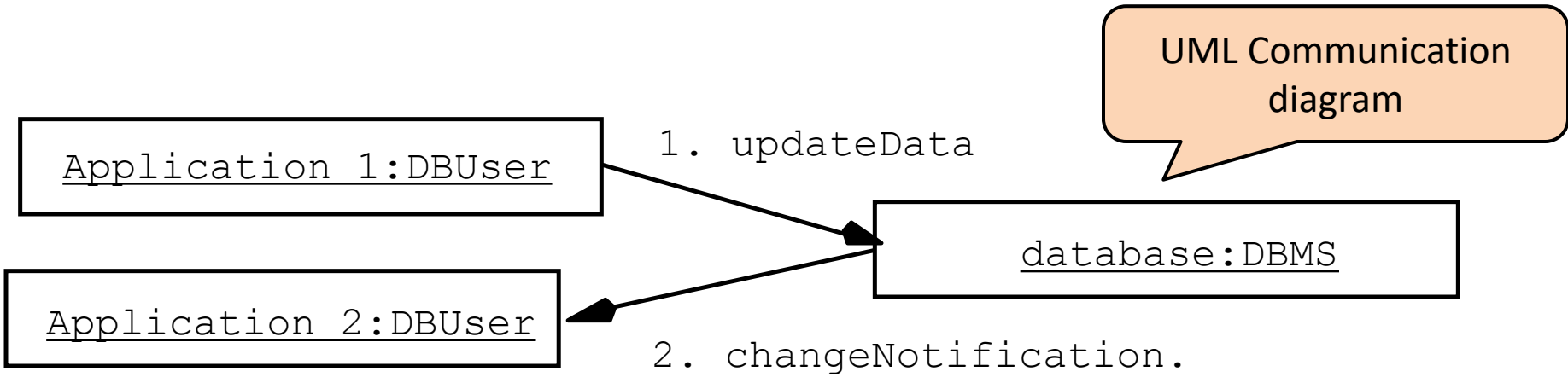


Design goals for client/Server architectures

Portability	Server runs on many operating systems and many networking environments
Transparency	Server might itself be distributed, but provides a single "logical" service to the user
High Performance	Client optimized for interactive display-intensive tasks. Server optimized for CPU-intensive operations
Flexibility	the Client supports a variety of end devices (Smartphone, Laptop, wearable computer)
Reliability	Server should be able to survive client and communication problems

Problems with Client/Server architectures

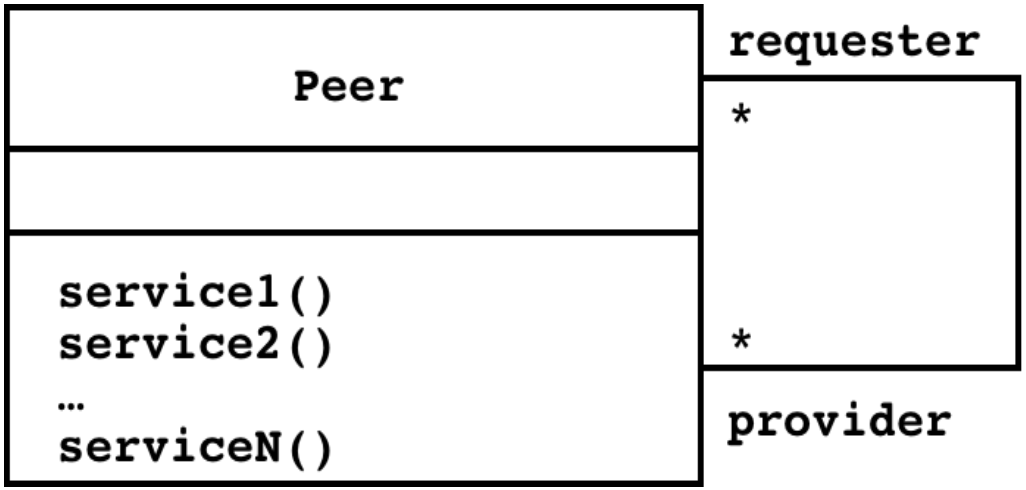
- Clients cannot communicate without the Server
- Scalability and availability can become a serious issue
- Example:
  - A database must process queries from application 1 and should be able to send notifications to application 2 when data in the database has changed



Peer-to-peer architectures

The Peer-to-Peer architectural style is a **generalization** of the Client/Server Architectural Style

- Clients can be servers and servers can be clients
- Introduction of a new abstraction: **Peer**

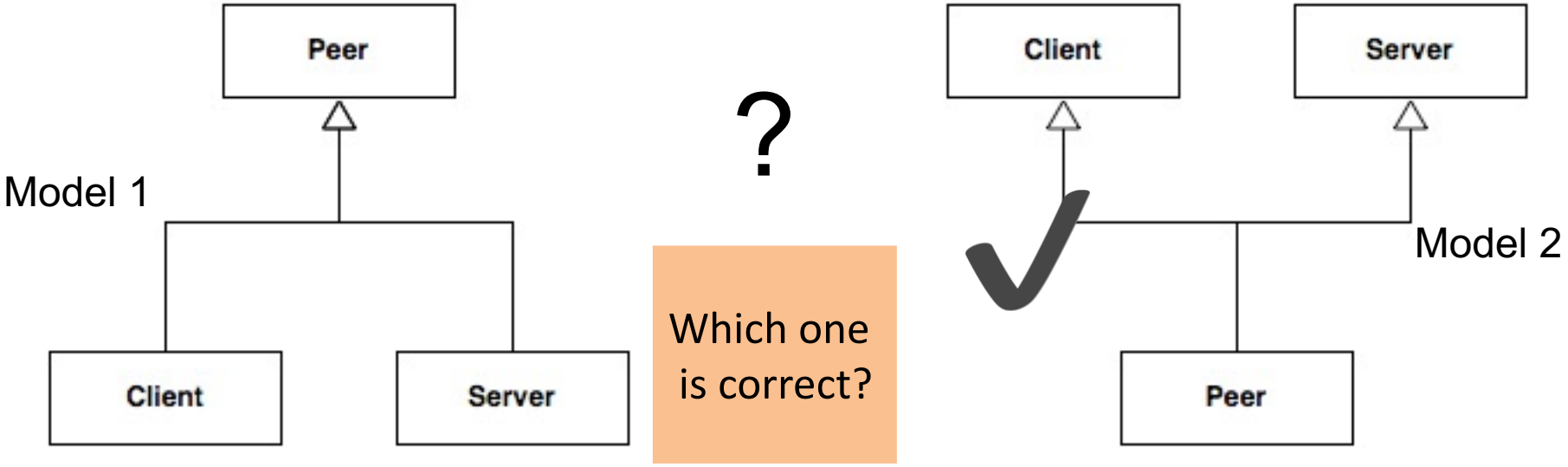


A small modeling exercise

- Problem statement:

  "Clients can be servers and servers can be clients"

- Interpretation 1: "A peer is either a client or a server"
- Interpration 2: "A peer is a client. A peer is a server"



Distributed systems: problem

- Building a software system as a monolithic application leads to problems with respect to **flexibility**, **maintainability** and **changeability**
- Better to design the system as a set of distributed systems running on different processes.
- However, this requires that the distributed components have to communicate with each other. A **communication mechanism** is needed

Distributed systems: solutions

- Solution A: Components implement the communication mechanism themselves.
  → severe dependencies and limitations
  Example: In a client-server architecture, the clients need to know the location of the server
- Solution B: Place the communication mechanism in a name server, decoupling clients and servers
  - The name server provides location transparency and hides the communication details

Client-Dispatcher-Server Pattern

- Solution C (Preferred):
  - Introduce a broker to decouple clients and servers
  - The communication between clients and servers is via remote service invocations
  - The broker is responsible for coordinating the communication

Broker Pattern

Outline of the talk

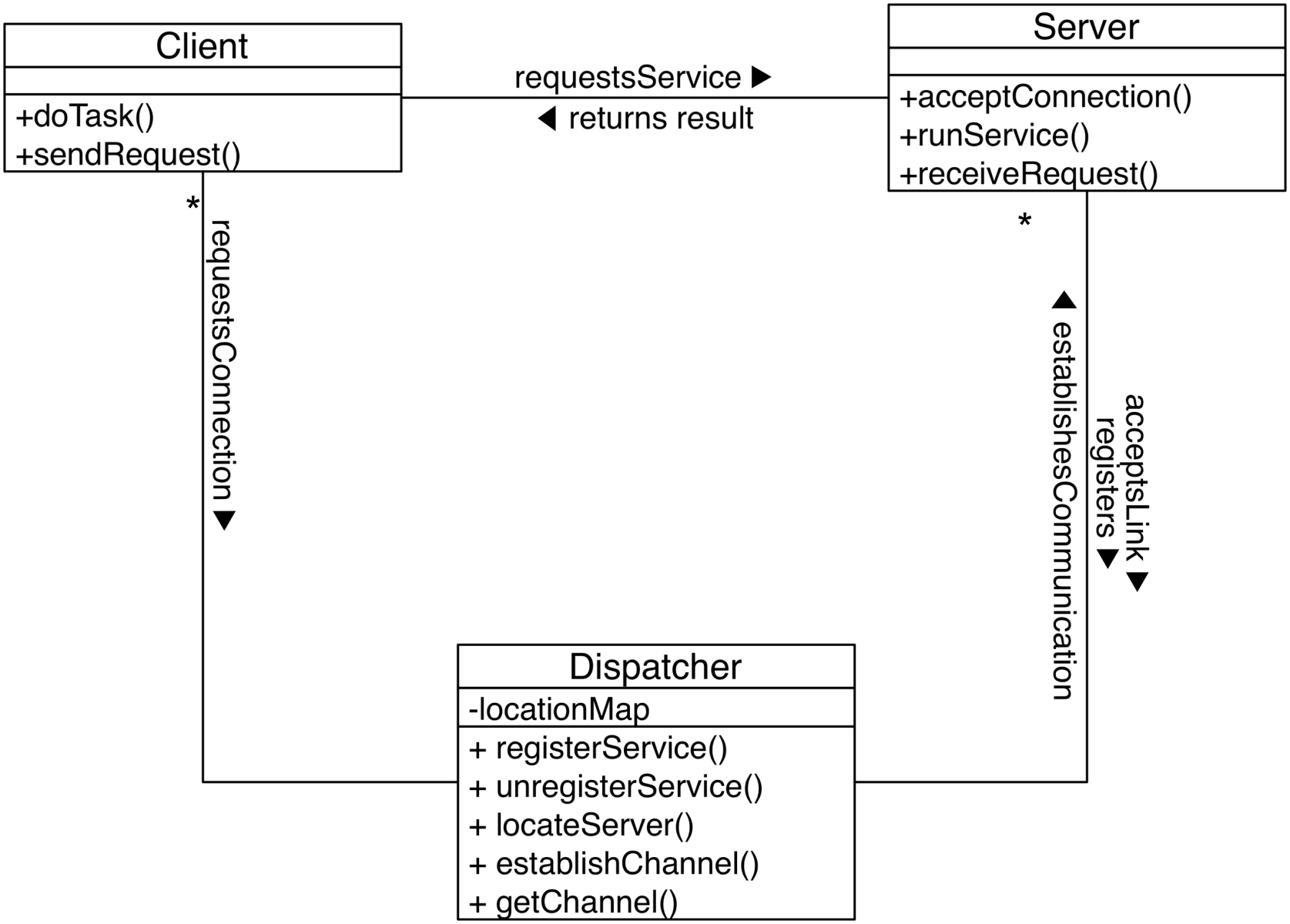
1. Motivation
2. **Client-Dispatcher-Server Pattern**
3. Broker Pattern

Client-dispatcher-server pattern

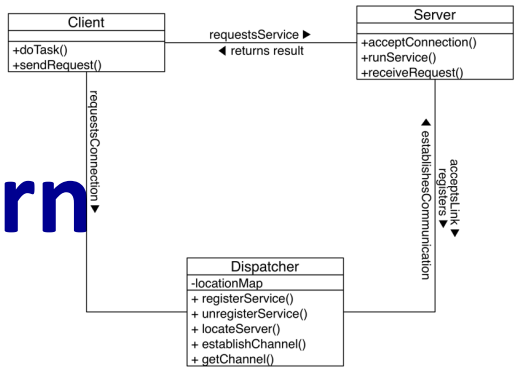
- When a client uses a remote server(s) distributed over a network it must ***establish a connection*** before the ***communication*** with the server works
- **Problem**: These two needs are not separated in many applications, causing unnecessary code complexity in service invocations. Better:
  - **Separate the core functionality** provided by the server **from the details** of the communication mechanism between client and server
  - Allow servers to dynamically **change their location** without impacting client code
- **Solution**: Insert a **Dispatcher** component between client and server that provides a name service:
  - Allows the client to refer to the server by names instead of physical locations (location transparency)
  - Establishes a channel between client and server, reducing the communication bottleneck inherent in the broker pattern.

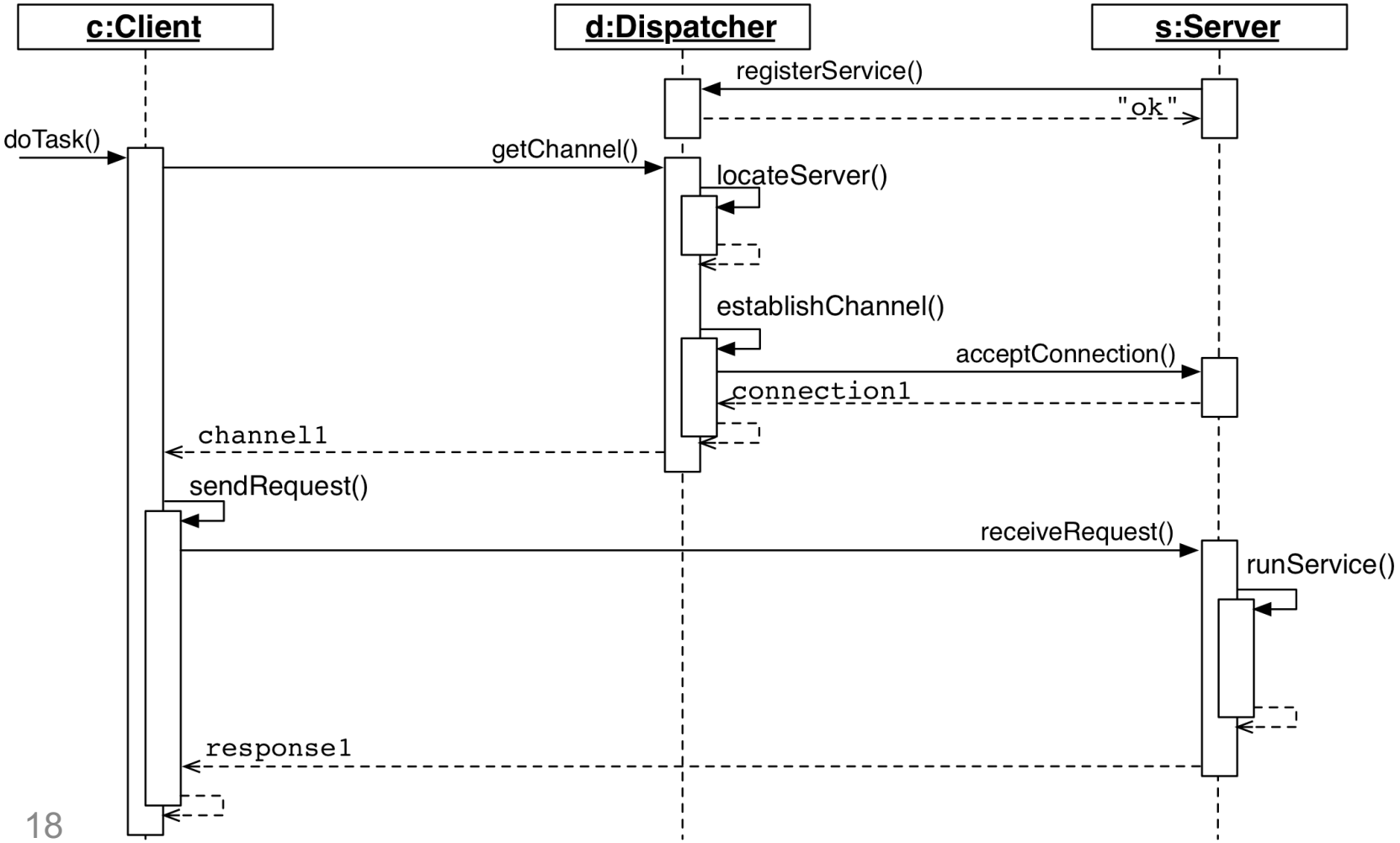
From client-server pattern….

… To client-dispatcher-server pattern



Typical scenario for the client-dispatcher-server pattern



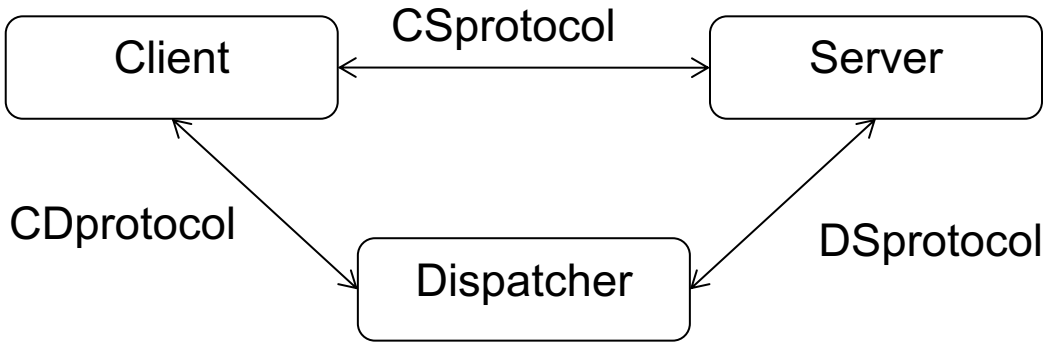


Steps to implement client-dispatcher-server

1. During system design (hardware-software mapping):
   identify the subsystems that act as clients and as servers
2. Decide which inter-process communication mechanisms are required
   (shared memory, sockets)
   - Communication between client and server
   - Communication between client and dispatcher
   - Communication between dispatcher and server

   Using a single communication mechanism decreases the complexity of the implementation but may not be efficient
3. Specify the interaction protocols between the components (set up the channel and define the transmission of data)
   - Protocol between client and server (CSprotocol)
   - Protocol between client and dispatcher (CDprotocol)
   - Protocol between dispatcher and server (DSprotocol).

Protocols in the client-dispatcher-server



- DSprotocol: Specifies how servers **register** with the dispatcher.
  Determines the activities needed to establish a **communication channel** between client and server
- CDprotocol: Specifies how a client must **look for** a particular server.
  Deals with communication **errors** (time out, server does not exist).
- CSprotocol: Specifies how clients and servers communicate with each other:
  1. The Client sends a message to the server using the communication channel established between them
  2. The Server receives the message, interprets it, and invokes one of its services. After the service is completed , the server sends a return message to the Client
  3. The Client extracts the result from the return message and continues its task.

More steps to implement the client-dispatcher-server

4. Decide on a naming scheme for the name server:
   - Introduce a naming scheme that uniquely define servers and are location transparent (URLs are ok, IP addresses are not)
5. Design and implement the dispatcher
   - Decide on how to implement the 3 protocols using available communication mechanisms, e.g. message based communication (sockets) or procedure calls (RMI).
     Example: use local procedure calls if client and dispatcher reside in the same address space)
   - Design and implement the following classes: Request, Response and Error Message
   - Design and implement a class Repository that maps server names to physical locations
   - Consider Performance issues
6. Implement the client and the server
   - Decide whether servers register with the dispatcher at system startup time or if servers can register und unregister dynamically

Example code: client

```
public class Client {
    public void doTask() {
        Service s;
        try {
            s = CDS.disp.locate("printSvc");
            s.service();
        } catch (NotFound n) { System.out.println("Not available."); }
        try {
            s = CDS.disp.locate("drawSvc");
            s.service();
        } catch (NotFound n) { System.out.println("Not available."); }
    }
}
```

Locate and execute the Print Service

Locate and execute the Drawing Service

Example code: dispatcher

```
public class Dispatcher {
    private HashMap<String, List<Service>> registry = new HashMap<String, List<Service>>();
    private Random rnd = new Random((new Date()).getTime());

    public void register(String svc, Service obj) {
        List<Service> l = registry.get(svc);
        if(l == null) {
            l = new ArrayList<Service>();
            registry.put(svc, l);
        }
        l.add(obj);
    }

    public Service locate(String svc) throws NotFound {
        List<Service> l = registry.get(svc);
        if(l == null) { throw new NotFound(); }
        if(l.size() == 0) { throw new NotFound(); }
        return l.get(rnd.nextInt(Integer.MAX_VALUE) % l.size());
    }
}
```

Registry for Services

Random Number Generator

Register a service with a name

Locate a service by name

Example code: server

```
public abstract class Service {
        protected String nameOfService;
        protected String nameOfServer;
        public Service(String svc, String srv) {
          nameOfService = svc;
          nameOfServer = srv;
          CDS.disp.register(nameOfService, this);
        }
        public abstract void service();
}

public class PrintService extends Service {
        public PrintService(String svc, String srv) {
          super(svc, srv);
        }
        @Override
        public void service() {
          System.out.println("Service " + nameOfService + " by " + nameOfServer);
          // Here the service code would be implemented.
        }
}
```

Note that example code is a Server with the name Service with one service() !!!!

Constructor for Service

Service registers itself with the Dispatcher

Code for the Print Service

Here we make sure that the abstract method service() is overwritten

Example code: application example

```
public class CDS {
    public static Dispatcher disp = new Dispatcher();

    public static void main(String[] args) {
        Service s1 = new PrintService("printSvc", "srv1");
        Service s2 = new PrintService("printSvc", "srv2");
        Client client = new Client();
        client.doTask();
    }
}
```

Known uses of client dispatcher server

- Remote Procedure Call (RPC) implementation by Sun [Sun90]
  - A combination of the Client-Dispatcher-Server variants *Distributed Dispatchers* and *Client-Server-Dispatcher with communication managed by clients* (described in Buschmann et al. pp 331, pp332)
- CORBA (Common Object Request Broker Architecture) [Corba]
  - Uses the principles of the Client-Dispatcher-Server pattern for refining and instantiating the *Broker* pattern
- UDDI: Web service directory
  - UDDI holds directory of web services
  - Can be queried to receive description of web service (WSDL)
  - Does not establish connection between client and server

Outline of the talk

1. Motivation
2. Client-Dispatcher-Server Pattern
3. **Broker Pattern**

Design goals addressed by the broker pattern

Location Transparency

- A client wants to use a service independently of the location of the server that offers it

Programming Language Independence

- Clients and servers be written in different programming languages

Decoupling of clients and servers

- Low coupling is a good system design principle

Decoupling of services and communication mechanisms

- A service implementation should not be mixed with communication code

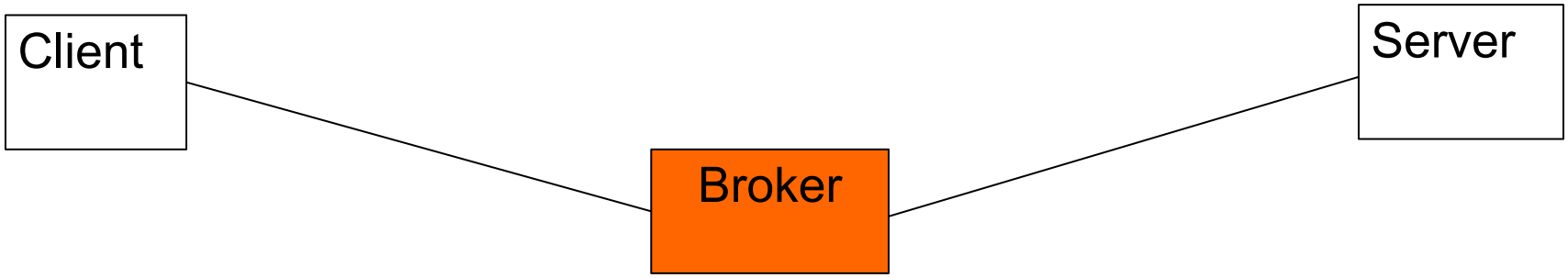
Runtime extensibility

- Being able to add, remove and exchange components at runtime

Components of the broker pattern

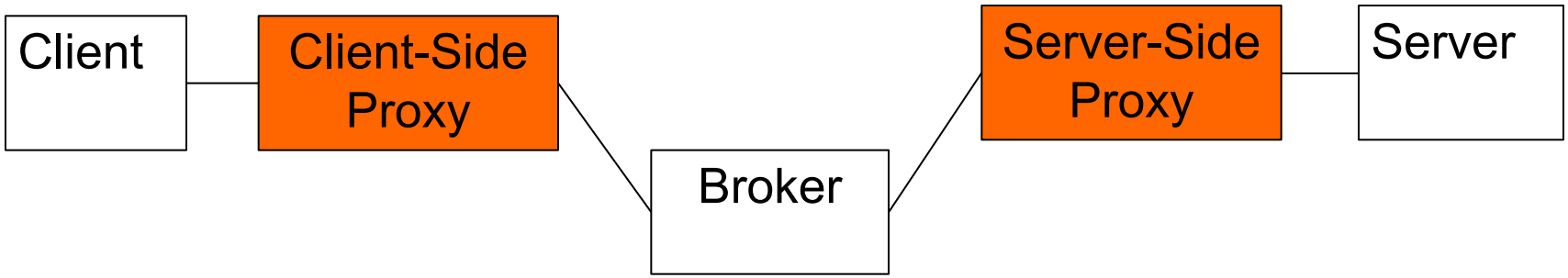
- Client: Any application that accesses the remote services of at least one server (Example: Web browser)
- Server: The Server class is responsible for providing remote services.
- There are two kinds of servers:
  - Provision of common services to **many domains** (Example: Web server)
  - **Specific functionality** for a specific application domain or task (Address book, Yellow pages, …)
- The Server exposes its functionality via an interface. This interface is made available either through
  - an interface definition language (IDL) such as JSON or
  - through a binary standard (works only for homogenous machines)

Components of the broker pattern



- Broker: Responsible for the **transmission of requests** from clients to servers as well as the **transmission of responses** and **exceptions** from the server back to the client
  - Local broker: A broker running on the same machine as the client
  - Remote broker: A broker server running on a remote machine (Example: Name server, Internet gateway).

Components of the broker pattern



- Client-Side Proxy: A layer between Client and Broker
- Server-Side Proxy: A layer between Broker and Server

Client-side proxy

- Makes the remote object (in the Server) **appear as** a local one
  - **Translates** the object model specified by the broker into an object model of the programming language used to implement the client
- **Hides** the inter-process communication **details** used for message transfer between client and broker
- Provides **marshalling** of parameters and results

Server-side proxy

- **Receives** requests from the broker
- **Hide** the inter-process communication details used for message transfer between broker and server
- Provides **unmarshalling** of parameters and results
- Calls the services in the server

Marshalling and unmarshalling

Marshalling

- Transforms the state and methods of an object to a byte stream suitable for transmission across a network. Synonym: **Serialization**

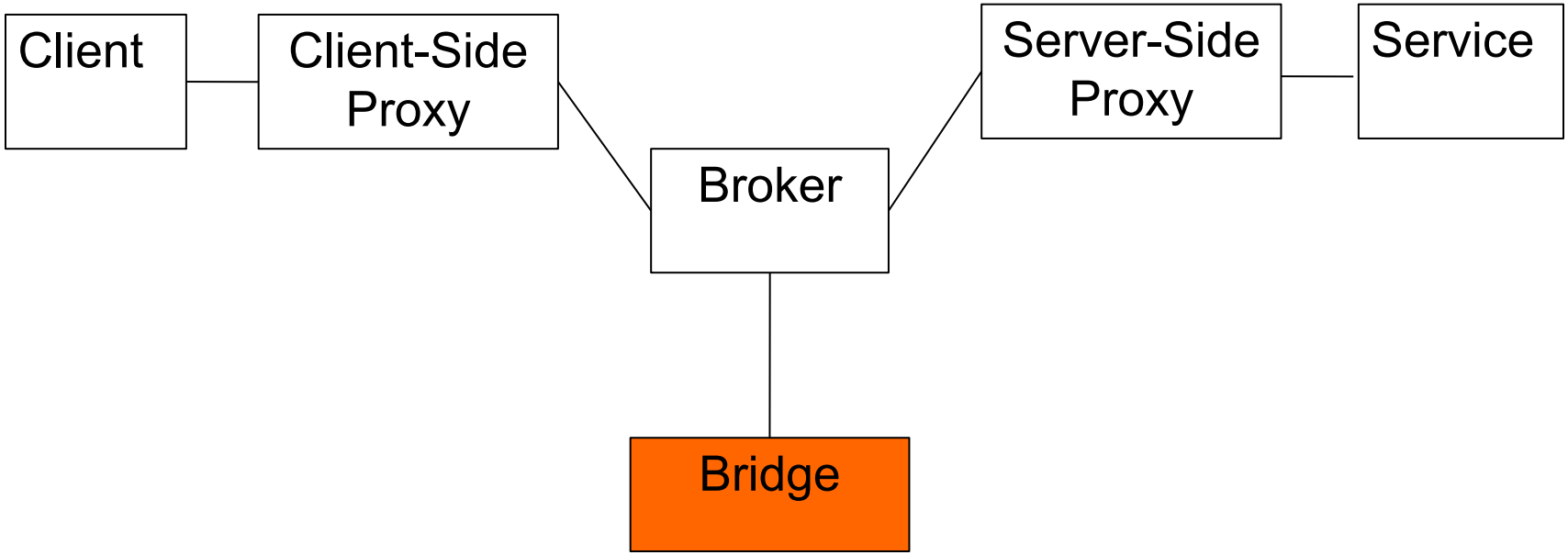
Unmarshalling

- The opposite of marshalling: Converting a byte stream back into the state and methods of an object. Synonym: **Deserialization**

Why do we need marshalling and unmarshalling?

- Used when the state of an object must be moved from one program on one machine to another program on another **heterogeneous** machine
- Typically used in implementations of remote procedure call mechanisms (RPC) and remote method invocation mechanisms (RMI, ...).
- Big Endian vs Small Endian machines
- Big-endian machines store the **most significant byte first**
- Llittle-endian machine stores the **least significant byte first**

Components of the broker pattern



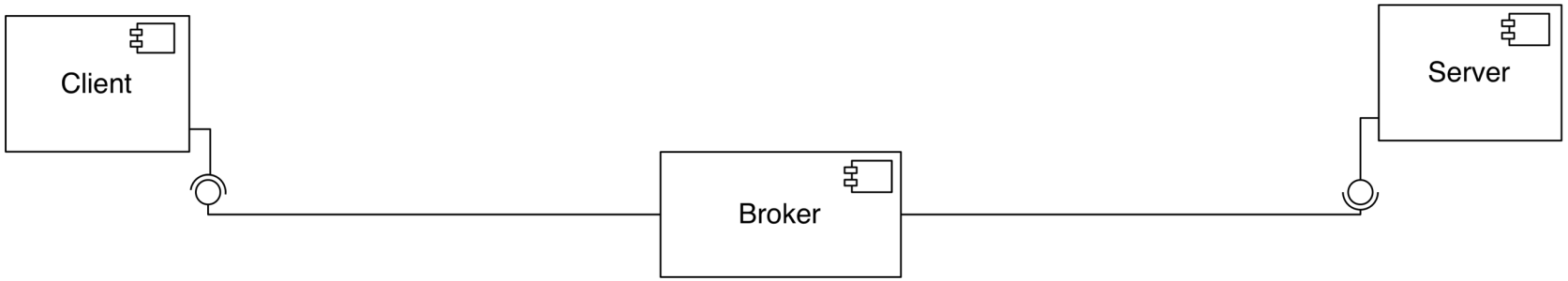
- **Bridge**: A component that hides implementation details when two Brokers have to interoperate in a heterogeneous environment.

Requirements realized by the Broker Pattern

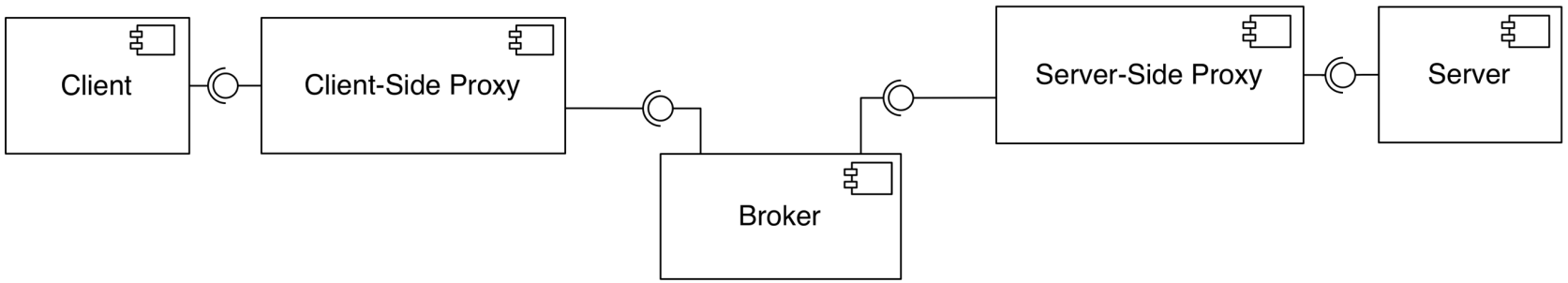
- Functional Requirements:
  - FR1: Client needs to find the Service
  - FR2: Clients uses Service

- Nonfunctional Requirements:
  - NFR1: Client invokes remote procedure as if it is local
    - Instead of using message based communication
  - NFR2: Client & Server can reside on heterogeneous Machines

Functional requirement FR1: client needs to find the service-> broker

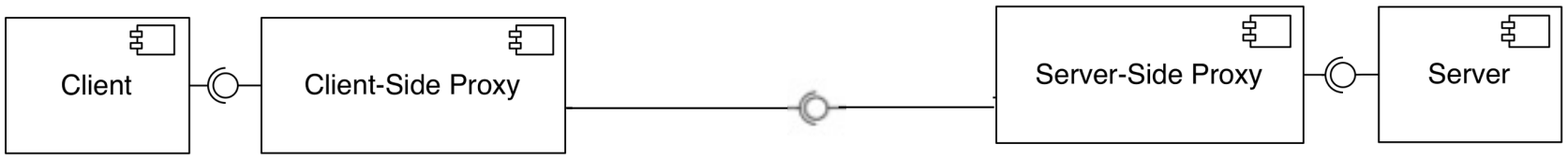


Nonfunctional requirement NFR1: client invokes remote procedure as if it is local



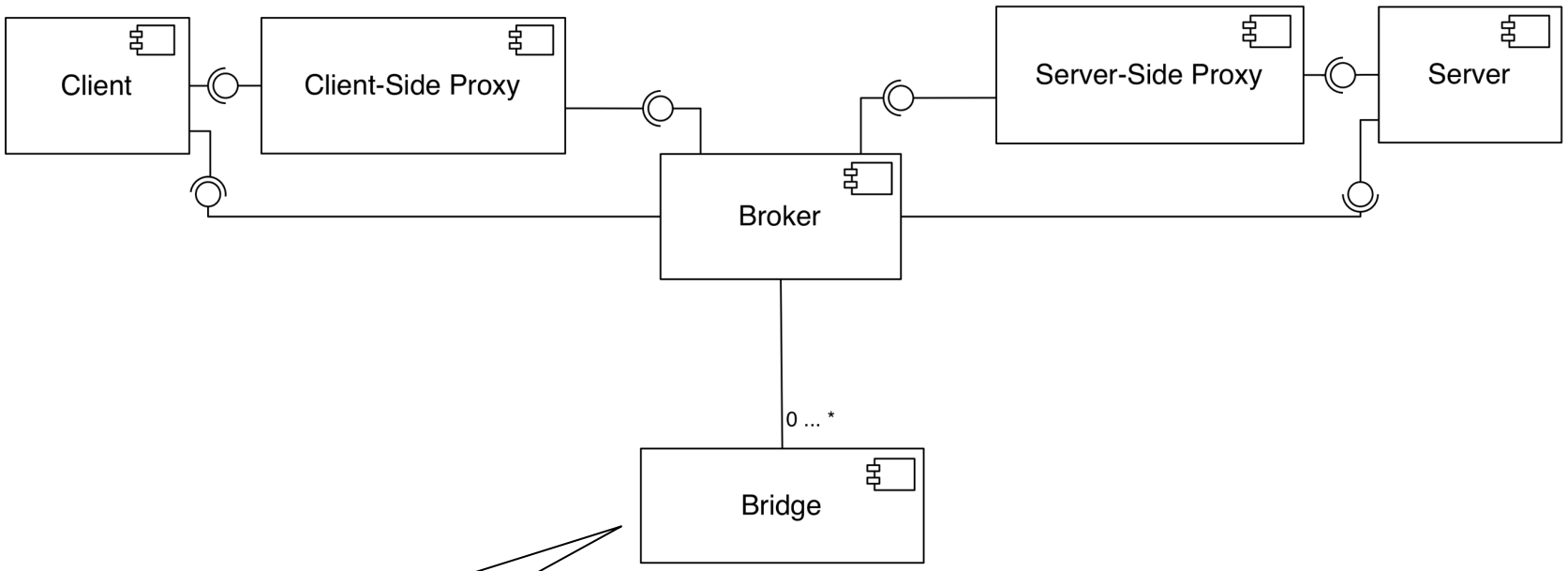
Functional requirement FR2:

clients uses service (invoke remote functionality)



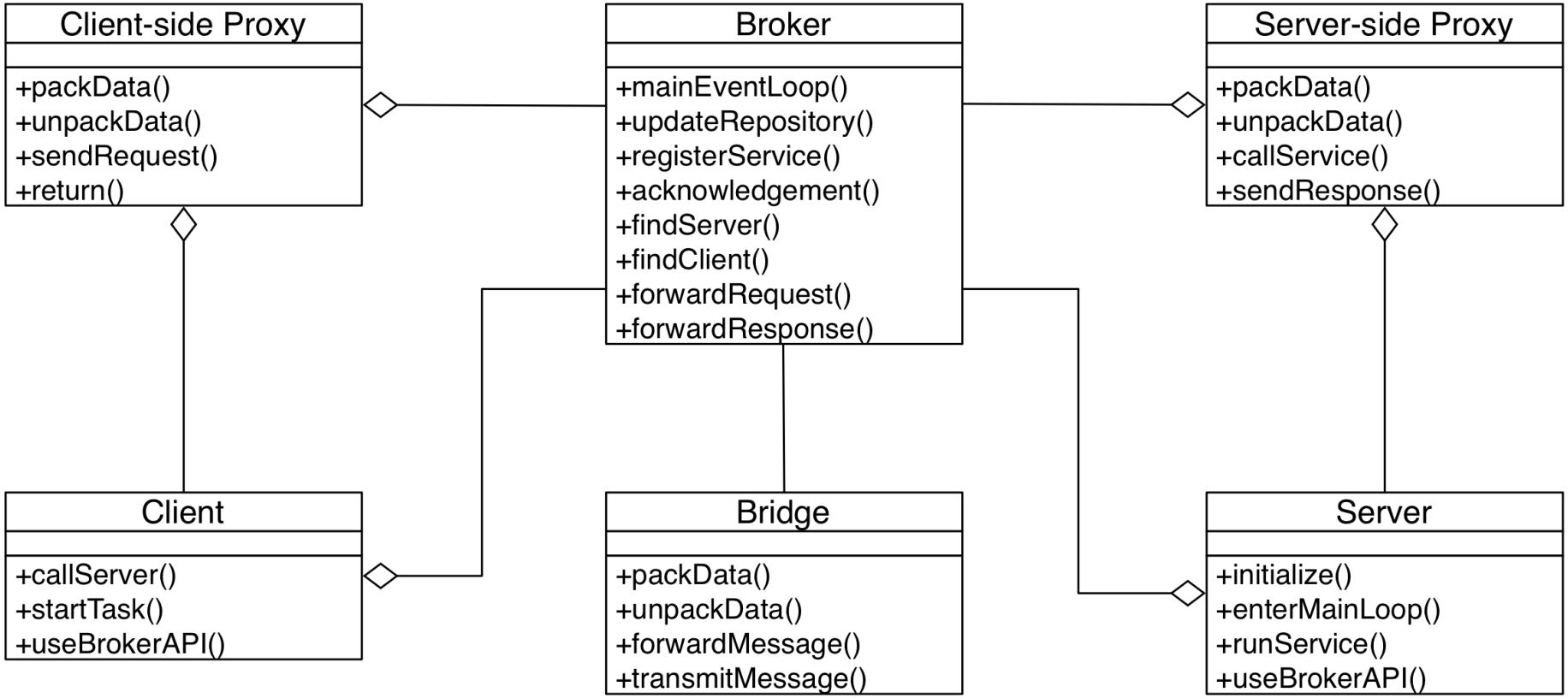
Nonfunctional requirement NFR2:

client & server can reside on heterogeneous machines

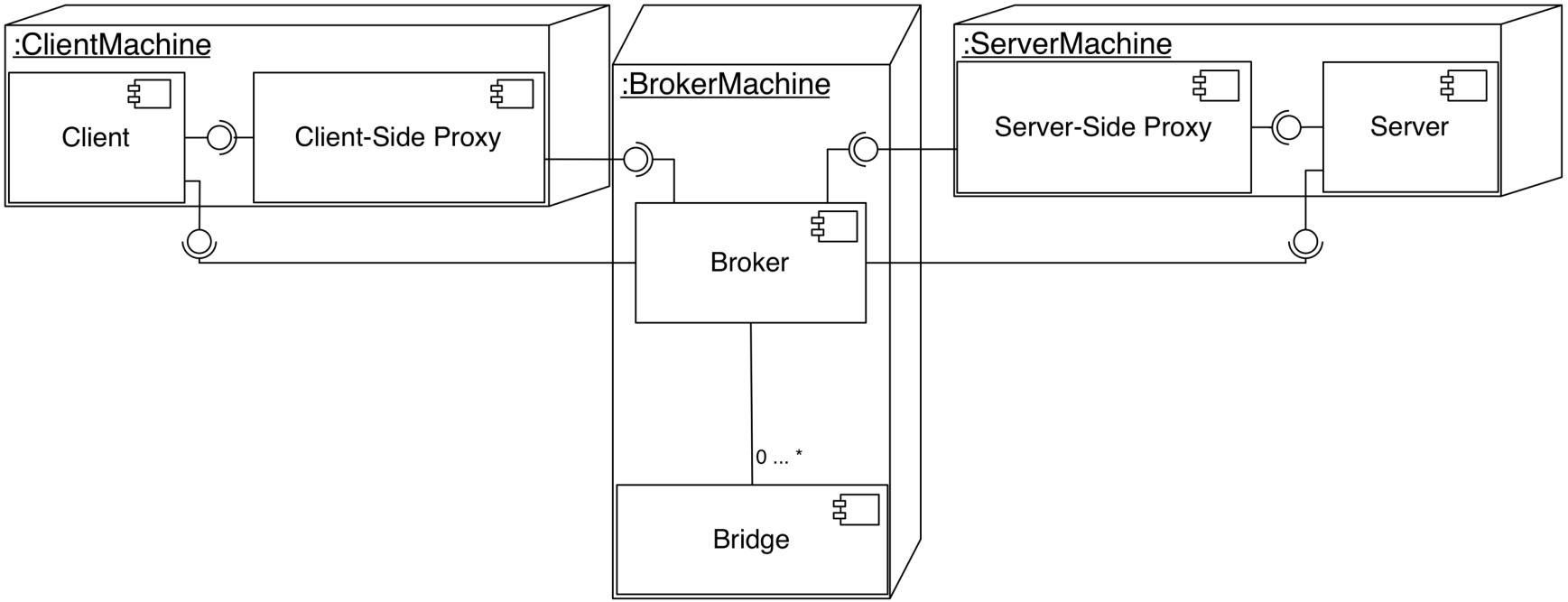


Bridge is the name of a component
(not be confused with the Bridge Pattern)

Broker pattern: UML class diagram



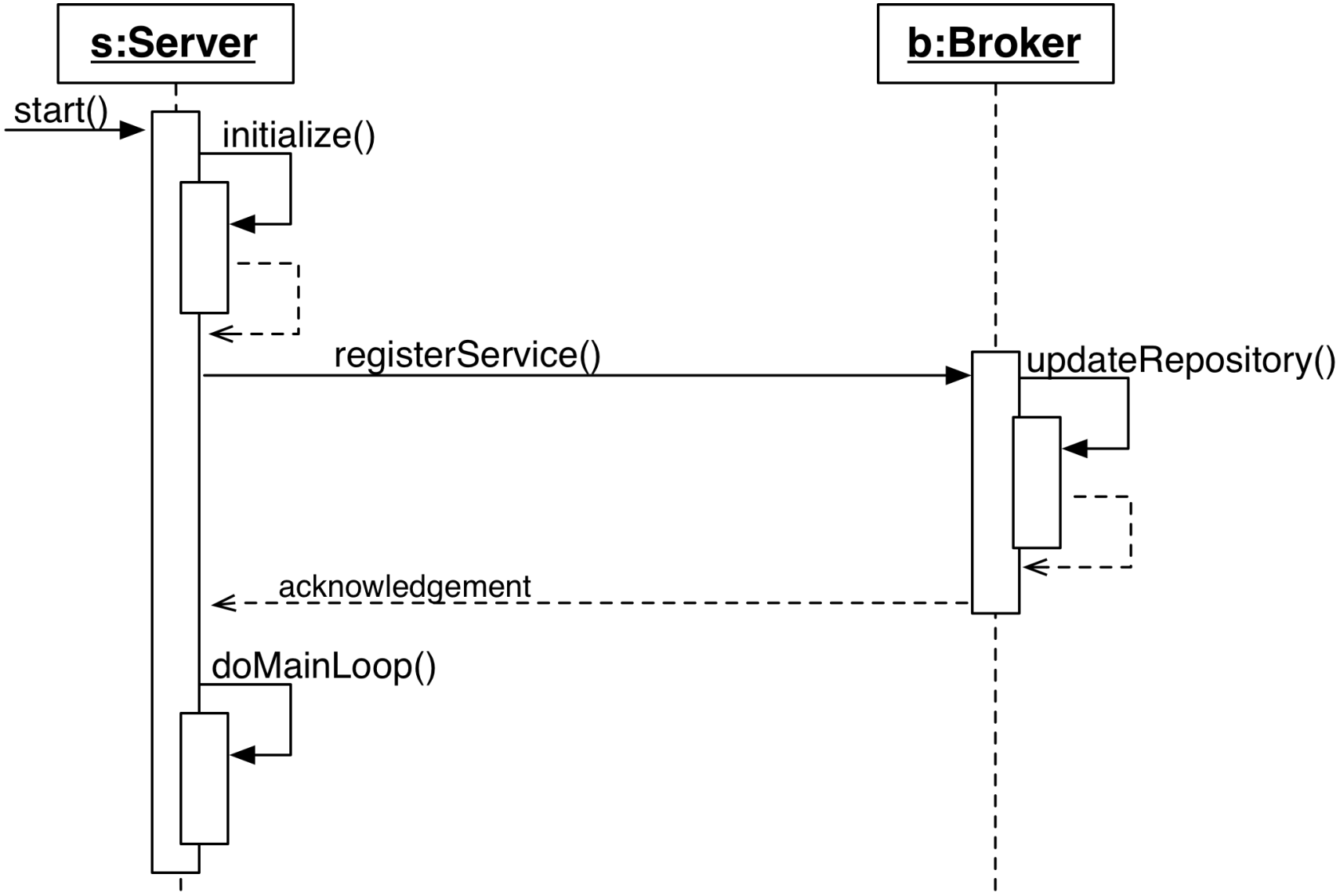
Broker pattern: UML deployment diagram



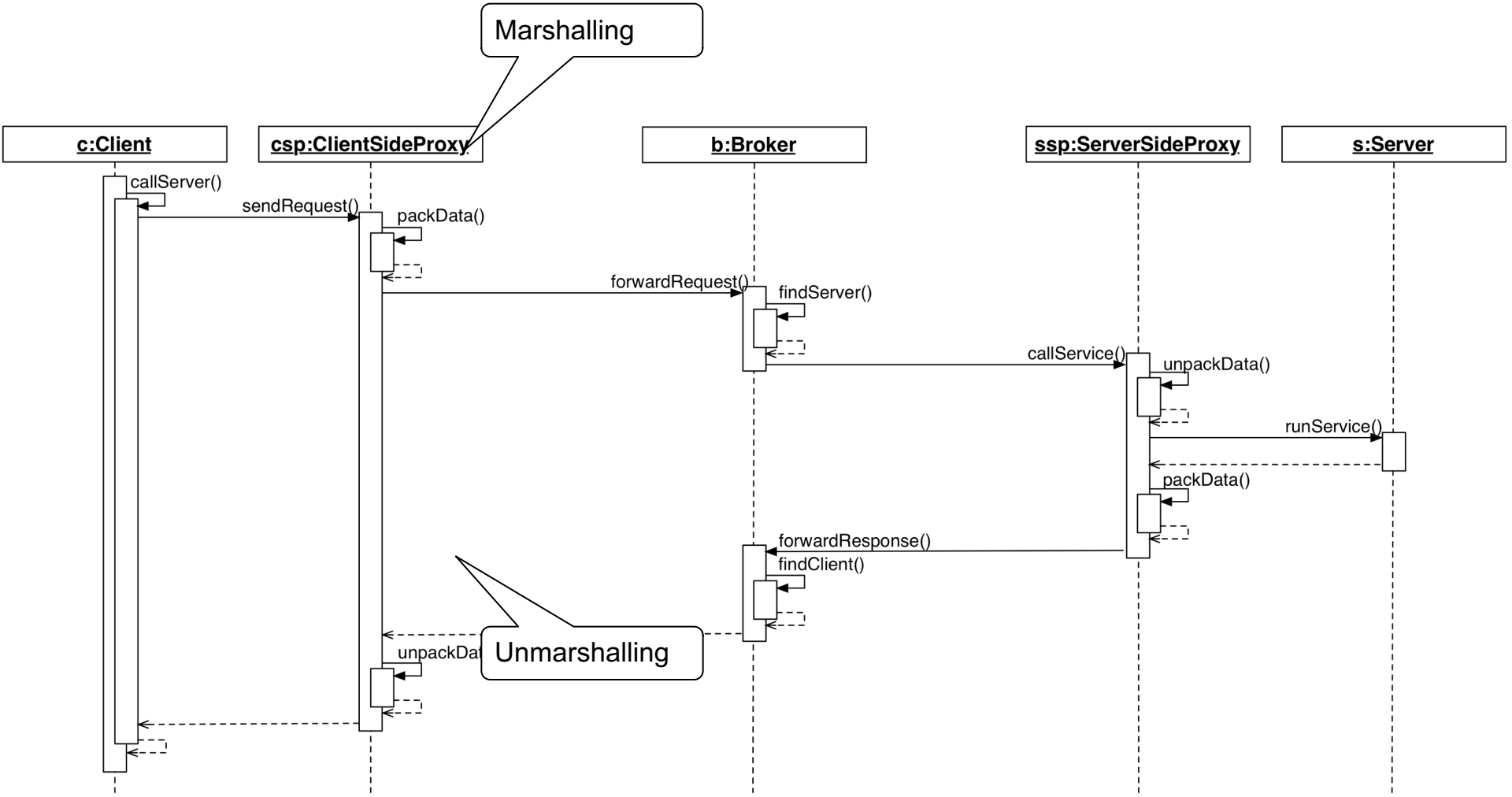
Typical scenarios in a distributed system using a broker pattern

- Scenario I: The Server component registers itself with the Broker Component
- Scenario II: The Client component sends a request to a Server Component
- Scenario III: Different Brokers interact via a Bridge component

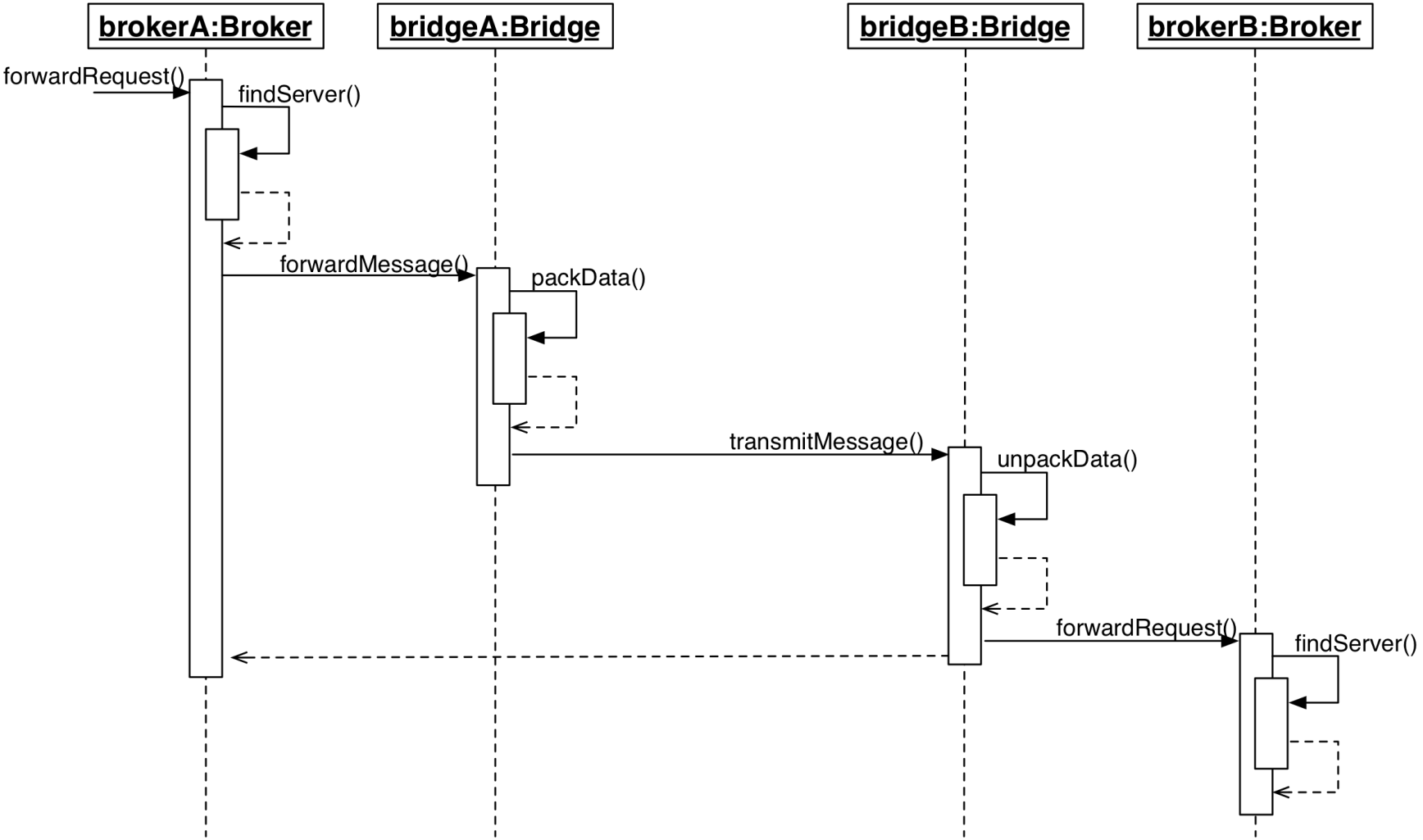
Scenario I: server registers with the broker



Scenario II: client sends a request to the server



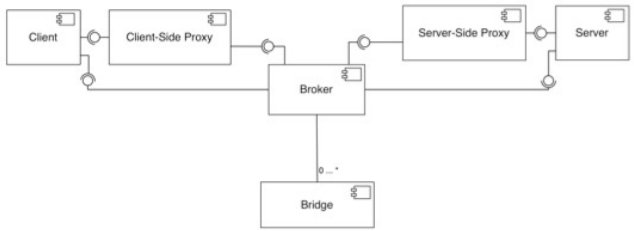
Scenario III: Interaction between Different Brokers



Steps to realize a broker pattern

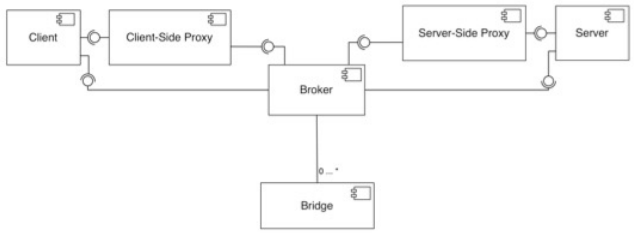
1. Define the Object Model
   - Define client and server objects. The state of the server objects should be private. Clients may change or read the server's state only by passing requests to the broker
2. Decide the level of interoperability
   - Decide whether to support only interoperability between binary compatible machines or the use of an interface definition language
3. Specify the API of the Broker component
   - Specify the services offered by the Broker for interprocess communication between the clients and the servers
4. Use proxy objects for client and server accesses
   - Design the client-side and server-side interfaces for the proxies
5. Design the broker component in parallel with steps 3 and 4
   - Implement the interface access routines for the proxies
6. Define the interface
   - Define the interface objects with an existing interface definition language (IDL)
   - Or define your own IDL and build your own IDL compiler.

Example code: client



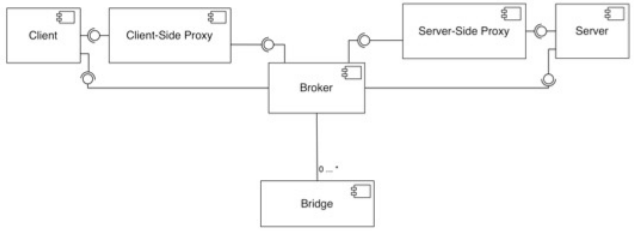

```
public class Client {

    protected ClientProxy Proxy = null;
    protected String ServiceName = null;
    protected String ServerName = null;
    protected String ClientName = null;

    public Client(String clientName, ClientProxy proxy, String serverName, String
    serviceName) {
      [...]
    }

    * Initiate server call via ClientProxy
    public void callServer() {
      System.out.println(ClientName + ": Calling service '" + ServiceName + "' on server '"
      +     ServerName + "'");
      String result = Proxy.sendRequest(ServerName, ServiceName, "foo");
      System.out.println(ClientName + ": Result was " + result);
    }
}
```

Example code: client proxy



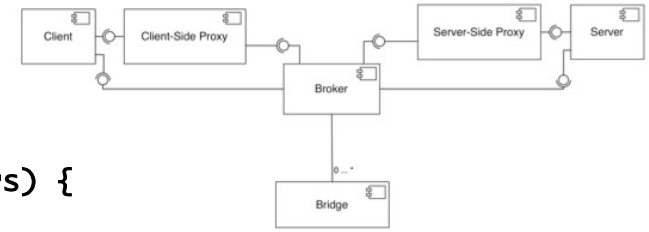

```
public class ClientProxy {

    * forwards service request from client to broker and does marshalling/ unmarshalling
    public String sendRequest(String serverName, String serviceName, String parameters) {
      // pack data
      String packedData = packData(parameters);

      // Execute service
      String result = Broker.forwardRequest(serverName, serviceName, "foo");

      // Unpack data and return
      String unpackedData = unpackData(result);

      return unpackedData;
    }
}
```

Example code: server



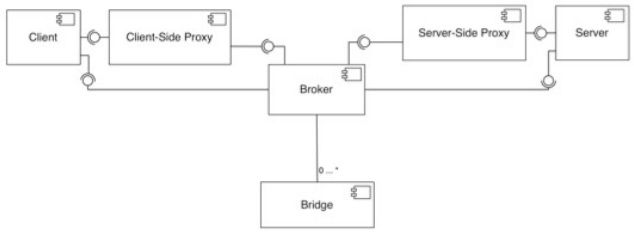

```
* Executes a service on this server
public String runService(String serviceName, String parameters) {
    // Actual service computation should be done here
    // Distinction between different services based on service name also
    System.out.println("Server '" + ServerName + "': Running service '" + serviceName + "'
    with   parameters '" + parameters + "'");
    String result = "Result of Server '" + ServerName + "' running service '" + serviceName
    + "' with parameters '" + parameters + "'";
    return result;
}

* Set a broker for this server and register all services of this server
public void setBroker(Broker broker) {
    // Deregister on old broker
    [...]
    this.Broker = broker;

    // Register all services on new broker
    for (String service : ServiceNames) {
    Broker.registerService(ServerName, service, Proxy);
}
```

Example code: server proxy



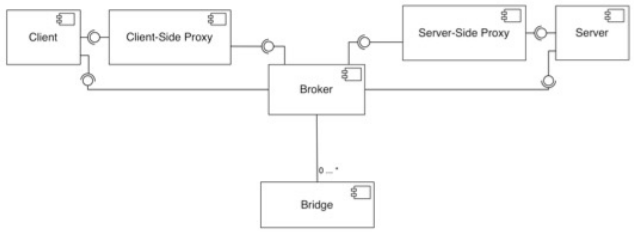

```
* Forward service call to server
public String callService(String serviceName, String parameters) {
    // Unpack data
    String unpackedParameters = unpackData(parameters);

    // Forward to server
    String result = Server.runService(serviceName, unpackedParameters);

    // Pack data
    String packedResult = packData(result);

    return packedResult;
}
```

Example code: broker



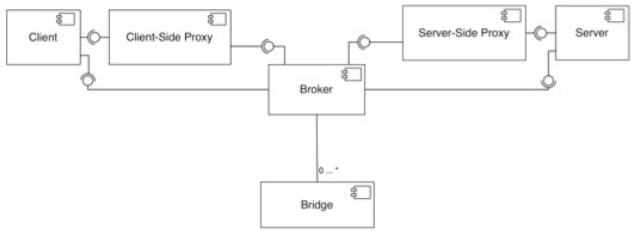

```
* Allows to register a service at this broker
public boolean registerService(String serverName, String serviceName, ServerProxy
    serverProxy) {
    RegisteredService s = new RegisteredService(serverName, serviceName, serverProxy);
    if(!Services.contains(s)) {
     Services.add(s);
     System.out.println("Registered service " + s);
    }
return true;
}
```

Example code: broker II




```
* Forward request from client proxy to broker
public String forwardRequest(String serverName, String serviceName, String parameters) {
String result;
    // lookup service and call
    for (RegisteredService service : Services) {
     if(service.getServerName().equals(serverName) && service.getServiceName().equals
     (serviceName)) {
          ServerProxy proxy = service.getServerProxy();
          result = proxy.callService(serviceName, parameters);
          return result;
     }
    }

    System.out.println("No server-service combination found for request ServerName='" +
      serverName + "', ServiceName='" + serviceName + "', Parameters='" + parameters + "'");
    return null;
}
```

Example code: driver




```
public class Driver {
    public static void main(String[] args) {
      // Setup broker
      Broker broker = new Broker();
      // Setup server 1
      Server server = new Server("Language server");
      ServerProxy serverProxy = new ServerProxy(server);
      server.setBroker(broker);
      // Setup client 1
      ClientProxy clientProxy = new ClientProxy(broker);
      Client client = new Client("Client 1", clientProxy, "Language server", "English
      service");
      client.callServer();
      // Setup client 2
      ClientProxy clientProxy2 = new ClientProxy(broker);
      Client client2 = new Client("Client 2", clientProxy2, "Language server", "English
            service FOO");
      client2.callServer();
    }
}
```

Summary

1. Problem of **distributed systems** is that components have to communicate with each other: communication mechanism is needed

2. In **Peer-to-Peer** Architectures clients can be servers and servers can be clients

3. The **Client-Dispatcher-Server** Pattern places the communication mechanism in a name server, decoupling clients and servers

4. The **Broker** pattern introduces a broker to decouple clients and servers

Readings

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal

- Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996

F. Buschmann, K. Henney, D. C. Schmidt, 2007

- Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, John Wiley & Sons, 2007

Sun Microsystems, Inc.

- Sun OS Documentation Tools, Formatting Documents, March 1990

Jonathan Swift

- Gullivers Travel, 1727

Corba: http://www.corba.org/

Intel Endianness White Paper, 2004