A04. *More on Architecture Patterns*

2014

Sungwon Kang

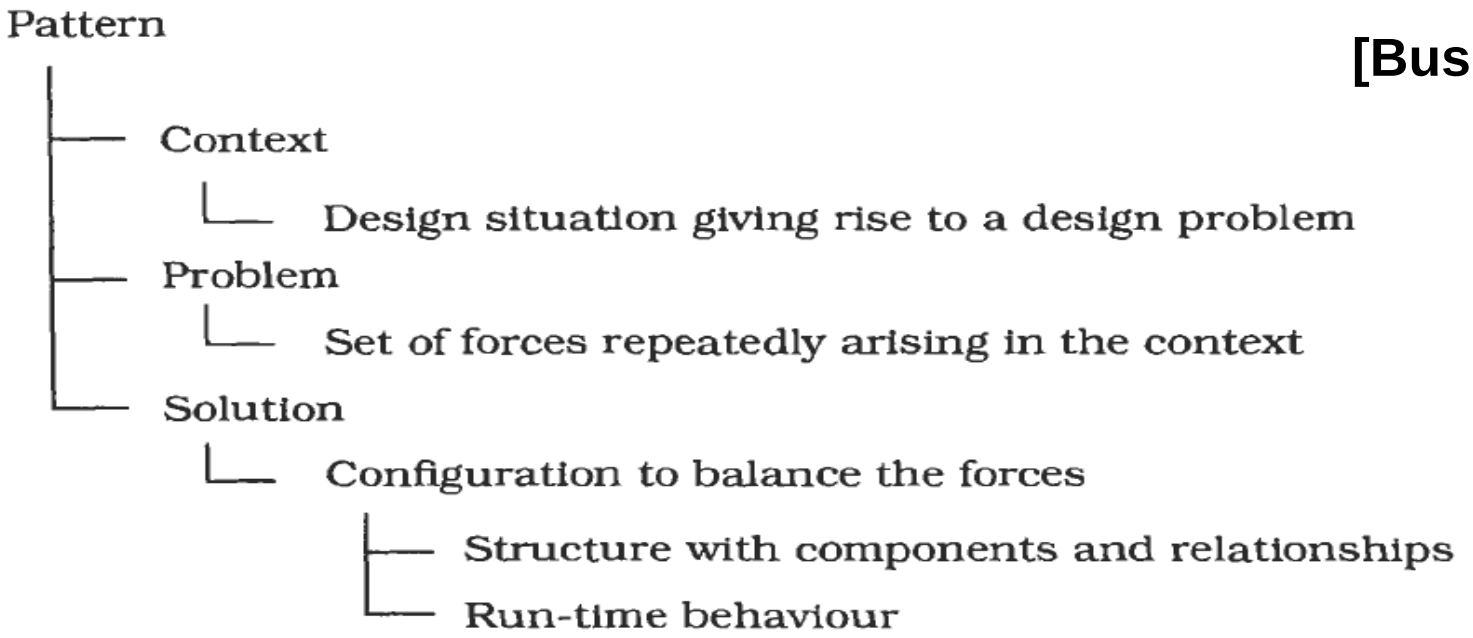
Table of Contents

1. Architecture Pattern Presentation

Layers Architecture Pattern Example

Acknowledgment Much of the following content is based on [Buschmann 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Wiley, 1996.

How Is a Pattern Presented?

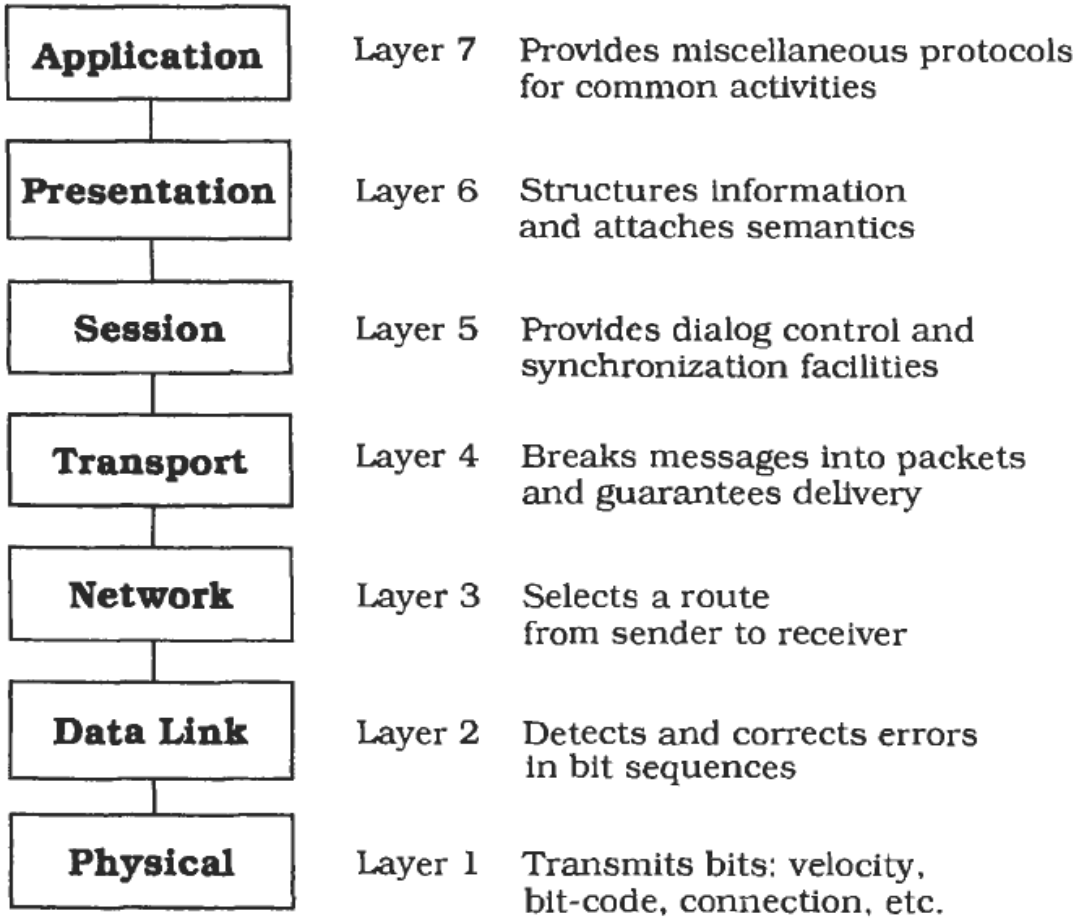


[Buschmann 96]

- Different from Architecture Style
  - No fine distinction of types of elements
  - No focus on constraints
  - Recall "An architectural style is a specialization of element and relation types, together with a set of constraints on how they can be used"

Layers Architecture Pattern

Example

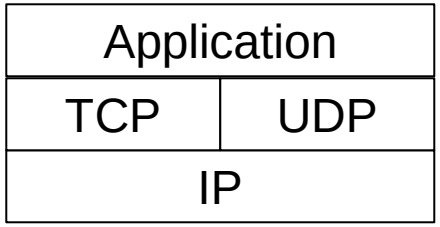


Context:

A large system that requires decomposition.

Problem (1/3)

- Dominant characteristic is a mix of low- and high-level issues where:
  - **Low-level issues**: hardware traps, sensor input, reading bits from a file or electrical signals from a wire.
  - **High-level issues**: user-visible functionality such as the interface of a multi-user 'dungeon' game or high-level policies such as telephone billing tariffs.
- Requests move from high to low level
- Answers to requests, data or notification about events, travel in the opposite direction.
- Often require some horizontal structuring that is orthogonal to their vertical subdivision.



Problem (2/3)

- The system specification describes the high-level tasks to some extent, and specifies the target platform.
- Portability to other platforms is desired.
- Several external boundaries of the system are specified a priori, such as a functional interface to which your system must adhere.
- The mapping of high-level tasks onto the platform is not straightforward, mostly because they are too complex to be implemented directly using services provided by the platform.

Problem (3/3)

Forces to consider:

- Late source code changes should not ripple through the system.
- Interfaces should be stable and may even be standardized.
- Parts of the system should be exchangeable.
- It may be necessary to build other systems at a later date with the same low-level issues as the system you are currently designing.
- Similar responsibilities should be grouped to help understandability and maintainability.
- Complex components need further decomposition.
- Crossing component boundaries may impede performance.
- The system will be built by a team of programmers, and work has to be subdivided along clear boundaries.

Solution

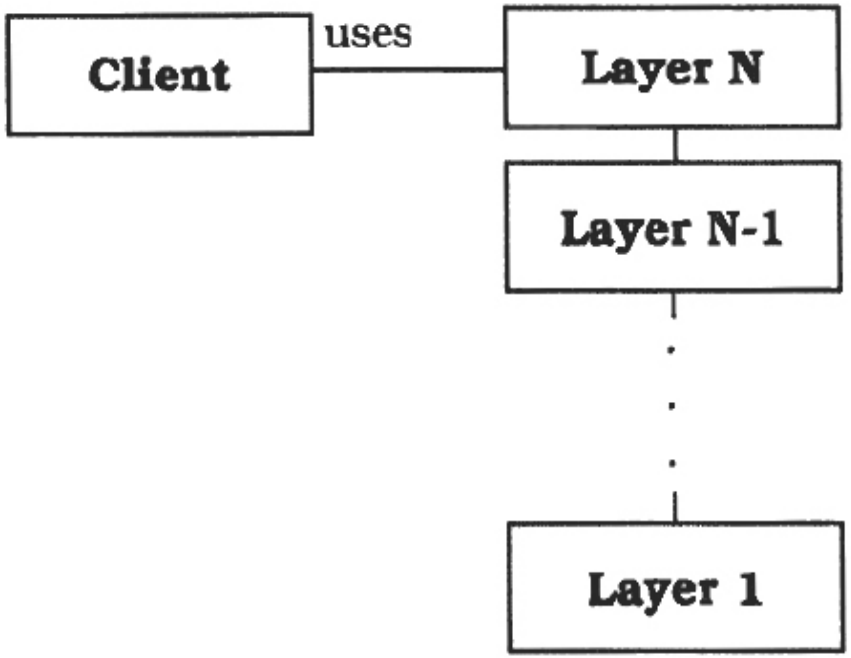
- Structure your system into an appropriate number of layers and place them on top of each other:
  - Start at the lowest level of abstraction - call it Layer 1.
  - Work your way up the abstraction ladder by putting Layer J on top of Layer J - 1 until you reach the top level of functionality - call it Layer N.
  - Most of the services that Layer J provides are composed of services provided by Layer J -1.
    - The services of each layer implement a strategy for combining the services of the layer below in a meaningful way.

  => Layer J's services may depend on other services in Layer J.

  => Services of Layer J are only used by Layer J + 1.

  => Each individual layer shields all lower layers from direct access by higher layers.

Solution – Structure (1/2)

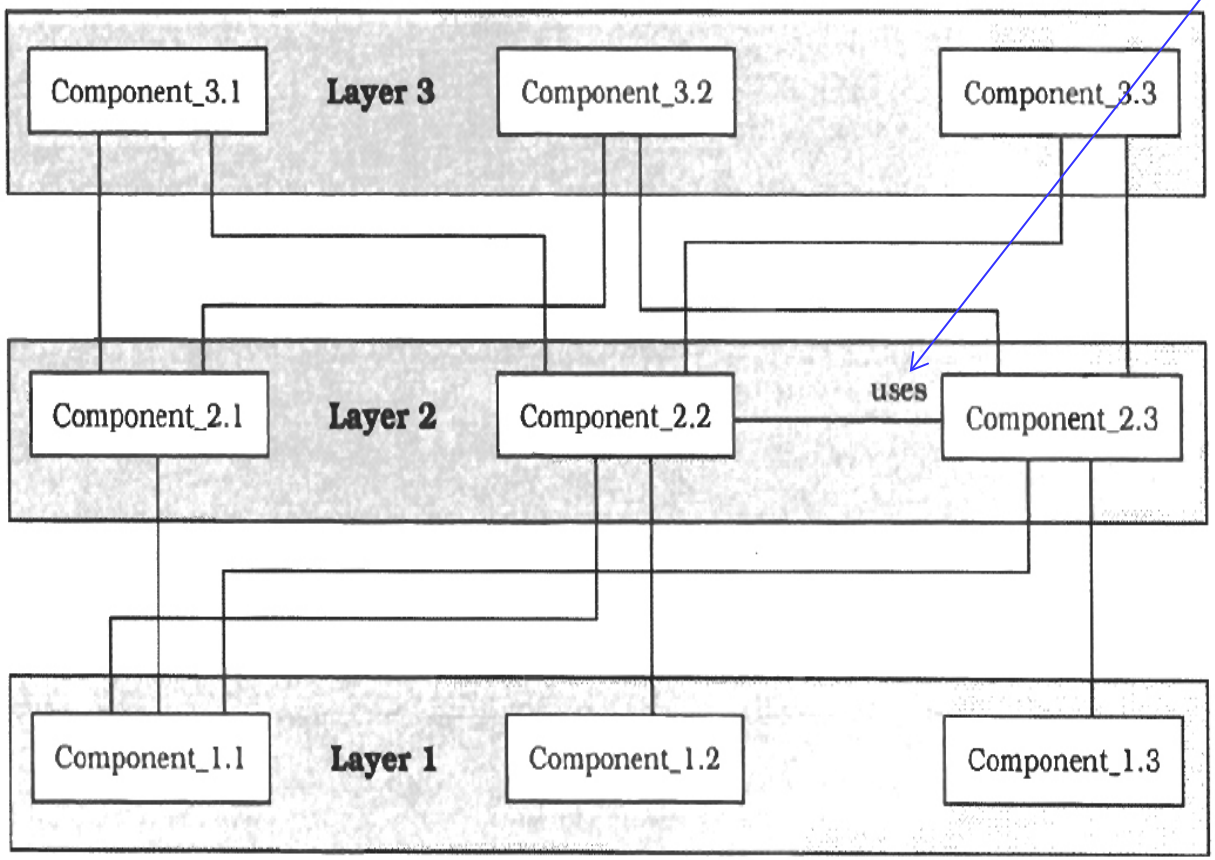


An individual layer can be described by the following CRC card:

Class Layer J	***Collaborator*** • Layer J-1
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	

Solution - Structure (2/2)

- Individual layers may be complex entities consisting of different components.



- In the middle layer two components interact.
- Components in different layers call each other directly
- Other designs shield each layer by incorporating a unified interface.
  - In such a design, Component_2.1 no longer calls Component_1.1 directly, but calls a Layer 1 interface object that forwards the request instead.

☛ Conceptual (or Logical) layering vs. Physical layering.

Solution - Dynamics (1/6)

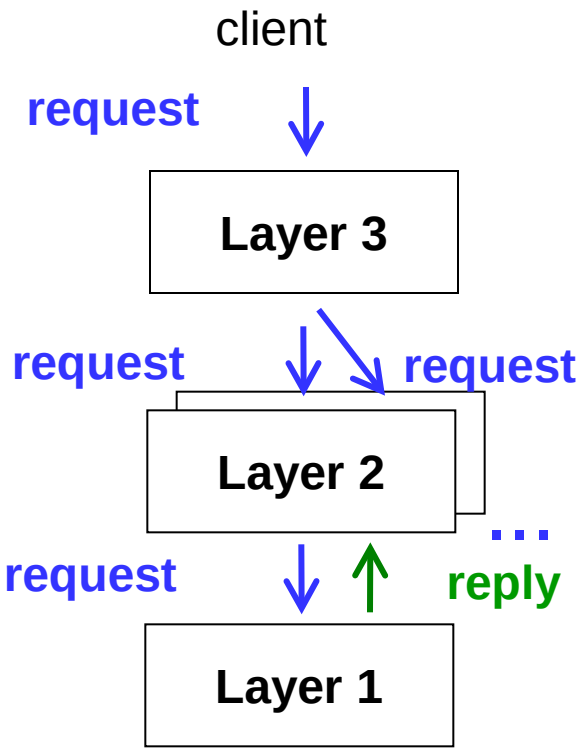
- The following scenarios are archetypes for the dynamic behavior of layered application:
  - In simple layered architectures you will only see Scenario 1
  - But most layered applications involve Scenarios I and 2.

Scenarios - Dynamics (2/6)

Scenario 1

- A client Issues a request to Layer N. Since Layer N cannot carry out the request on its own, it calls Layer **N** - **1**.

  In the process sending further requests to Layer N-2 and so on until Layer 1 is reached where the lowest-level services are finally performed.
- If necessary, replies to the different requests are passed back up from Layer 1 to Layer 2, from Layer 2 to Layer 3, and so on until the final reply arrives at Layer N.
- Layer J often translates a single request from Layer J+1 Into several requests to Layer J- 1.



Scenarios - Dynamics (3/6)

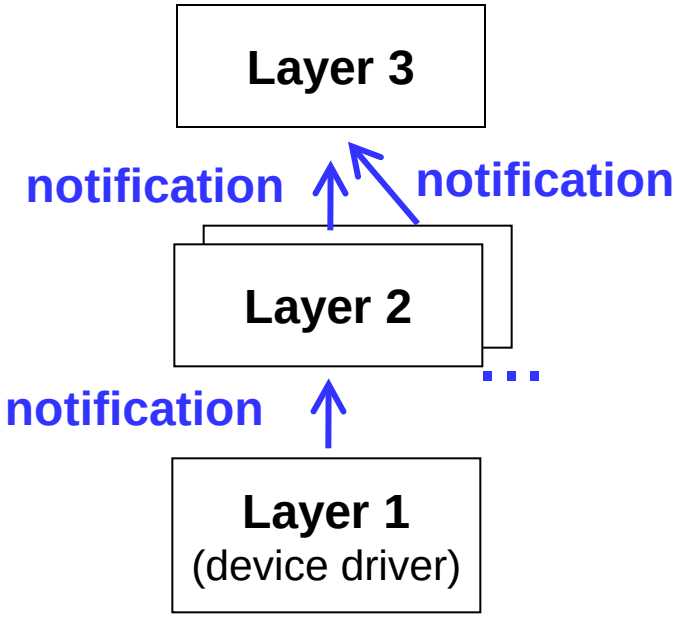
Scenario 2

- Illustrates bottom-up communication chain of actions starts at Layer 1.

Example

When a device driver detects input, it translates the input into an Internal format and reports it to Layer 2. which starts interpreting it, and so on.

- Data moves up through the layers until It arrives at the highest layer.
- Bottom-up calls can be termed 'notifications'.



Scenarios - Dynamics (4/6)

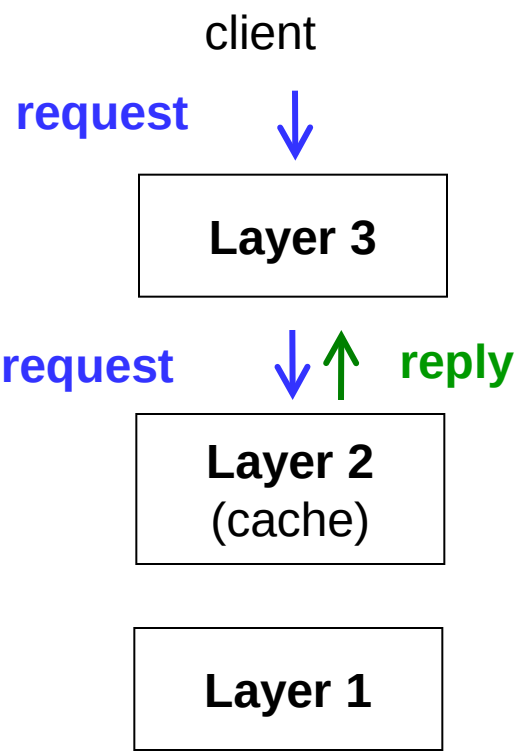
Scenario 3

- Requests only travel through a subset of the layers.
- A top-level request may only go to the next lower level N- 1 if this level can satisfy the request.

Example

- Level N- 1 acts as a cache, and a request from Level N can be satisfied without being sent all the way down to Layer 1 and from here to a remote server.

  => Such caching layers maintain state information, while layers that only forward requests are often stateless.



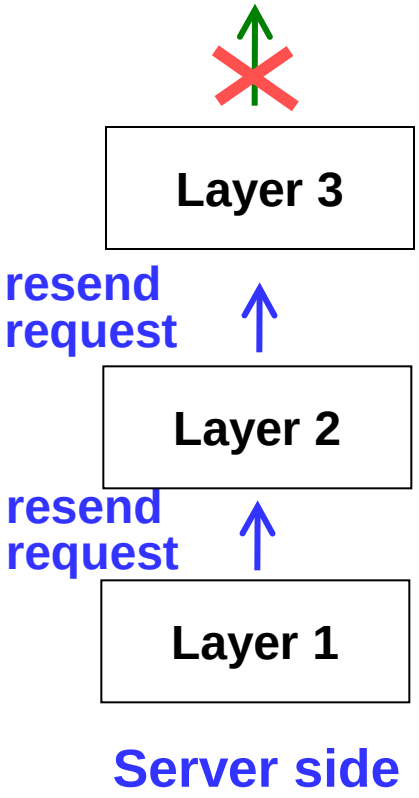
Scenarios - Dynamics (5/6)

Scenario 4

- Similar to Scenario 3.
- An event is detected in Layer 1, but stops at Layer 3 instead of traveling all the way up to Layer N.

Example In a communication protocol a resend request may arrive from an impatient client who requested data some time ago.

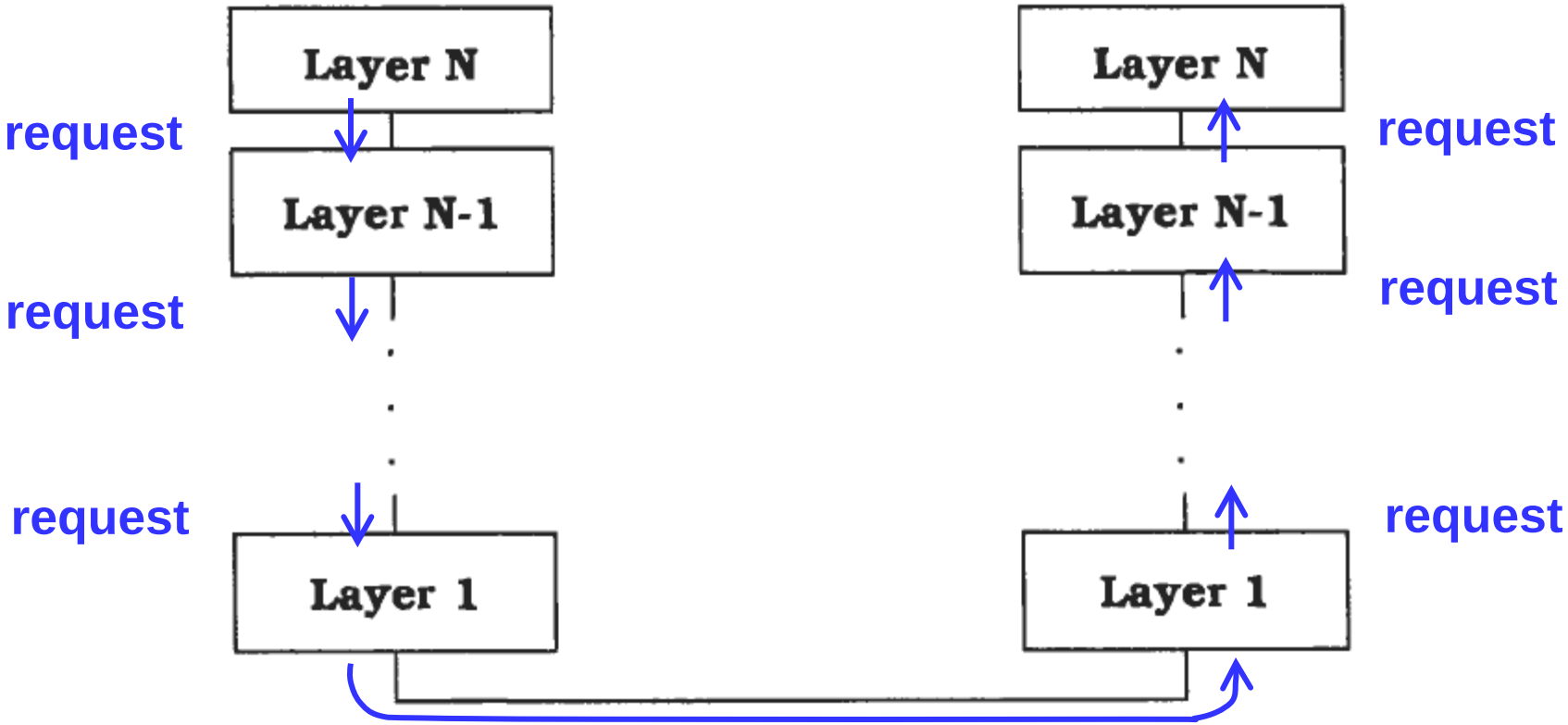
- In the meantime the server has already sent the answer, and the answer and the re-send request cross.
- In this case, Layer 3 of the server side may notice this and intercept the re-send request.



Scenarios - Dynamics (6/6)

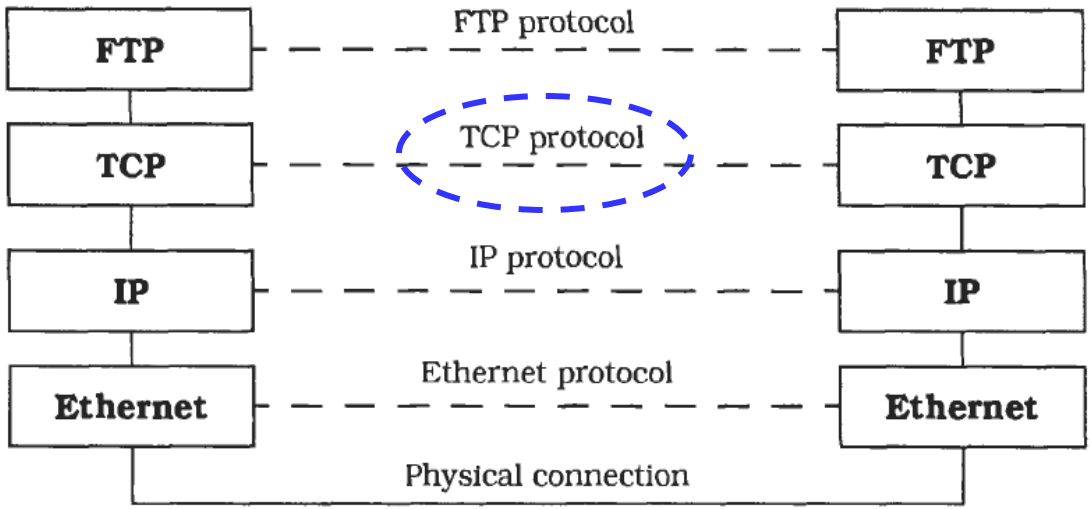
Scenario 5

- Involves two stacks of N layers communicating with each other.
- Well-known from communication protocols where the stacks are known as 'protocol stacks'.



Example Resolved (1/3)

- **TCP/IP**
  - Does not strictly conform to the OSI model
  - Consists of only four layers
  - TCP and IP constitute the middle layers, with the application at the top and the transport medium at the bottom.
- A typical configuration, that for the UNIX ftp utility:



Example Resolved (2/3)

- **TCP/IP**

- Corresponding layers communicate in a peer-to-peer fashion using a **virtual protocol**.

=> The two TCP entities send each other messages that follow a specific format.

- Every TCP implementation exports a fixed set of core functions but is free to offer more, for example to increase flexibility or performance.

  <= The vendors usually offer full implementations of the protocol suite.

- This looseness has no impact on the application developer because:

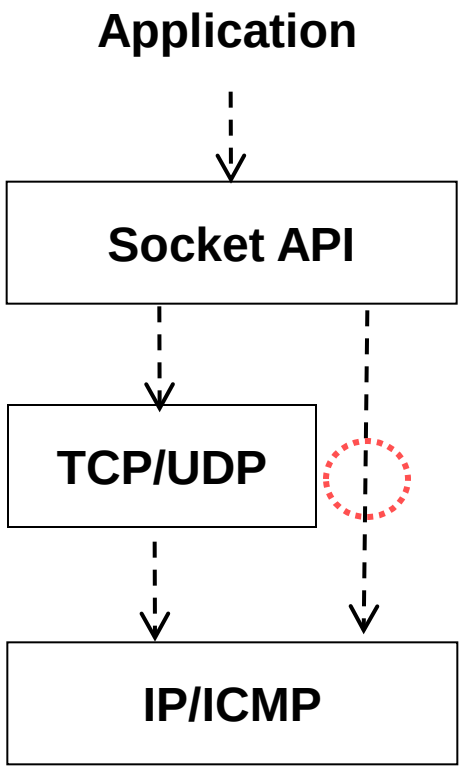
  (1) Different stacks understand each other because the virtual protocols are strictly obeyed.

  (2) Application developers use a layer on top of TCP. This upper layer has a fixed interface such as Sockets and TLI .

Example Resolved (3/3)

- The Socket implementation sits conceptually on top of TCP/UDP, but uses lower layers as well, for example IP and ICMP.
- This violation of strict layering principles is worthwhile to tune performance, and can be justified when all the communication layers from sockets to IP are built into the OS kernel.
- However, the behavior of the individual layers and the structure of the data packets flowing from layer to layer are much more rigidly defined **than** the functional interface.

  <= Different TCP/IP stacks must understand each other



Variants (1)

Relaxed Layered System

- Less restrictive about the relationship between layers.
- Each layer may use the services of all layers below it, not only of the next lower layer.
- A layer may also be partially opaque

=> some of its services are only visible to the next higher layer, while others are visible to all higher layers.

- The gain of flexibility and performance is paid for by a loss of maintainability.
  - Often seen in infrastructure systems such as UNIX OS and X Window System

  <= modified less often than application systems,

  and their performance is usually more important than their maintainability.

Variants (2)

Layering through Inheritance

- Lower layers are implemented as base classes.

  A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.

- Can be found in some object-oriented systems

- **Advantage**: higher layers can modify lower-layer services according to their needs.

- **Drawback**: inheritance relationship closely ties the higher layer to the lower layer.

  Example The data layout of a C++ base class changes, all subclasses must be recompiled. Such unintentional dependencies introduced by inheritance are also known as the ***fragile base class problem.***

Known Uses (1/3)

Virtual Machines.

- lower levels as a ***virtual machine*** that insulates higher levels from low-level details or varying hardware.

Example

- Java Virtual Machine (JVM)

API(Application Programming Interface)

- a layer that encapsulates lower layers
- usually a flat collection of function specifications

  Example C standard library operations like printf( ) above UNIX system calls

- **Advantage**: portability between different OSs, additional higher-level services such as output buffering or formatted output, less opportunities for errors and conceptual mismatches
- **Liability**: lower efficiency, perhaps more tightly-prescribed behavior

Known Uses (2/3)

Information Systems (IS)

- business software domain often use a two-layer architecture.
  - **Bottom layer**: a database that holds company-specific data.
  - **Top layer**: many applications work concurrently on top of database

Example

- Mainframe interactive systems, Client-Server systems

Disadvantage: tight coupling of user interface and data representation causes its share of problems

=> **Three layer architecture**: introduce the domain layer-which models the conceptual structure of the problem domain between them.

- **Four-layer architecture**: separate user interface and application

=> Presentation/Application logic/Domain layer/Database

Known Uses (3/3)

Windows NT

- structured according to the Microkernel pattern

- **NT** Executive component corresponds to the microkernel component

- **NT** Executive is a Relaxed Layered System

Layers:

- **System services**: the interface layer between the subsystems and the **NT** Executive.

- Resource management layer: contains the modules Object Manager, Security Reference Monitor, Process Manager, 1/0 Manager, Virtual Memory Manager and Local Procedure Calls.

- **Kernel**: takes care of basic functions such as interrupt and exception handling, multiprocessor synchronization, thread scheduling and thread dispatching.

- **HAL (Hardware Abstraction Layer**): hides hardware differences

- **Hardware:** Windows NT relaxes the principles of the Layers pattern because the Kernel and the 1/0 manager access the underlying hardware directly for reasons of efficiency.

Consequences

- **Benefits**
  - Reuse of layers
  - Support for standardization
  - Dependencies are kept local
  - Exchangeability

- **Liabilities**
  - Cascades of changing behavior
  - Lower efficiency
  - Unnecessary work
  - Difficulty of establishing the correct granularity of layers

Resources – Architecture Patterns

[Buschmann 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, WILEY, 1996.

[Schimidt 00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*

[Kircher 04] Michael Kircher and Prashant Jain, *Pattern-Oriented Software Architecture, Volume 3 - Patterns for Resource Management*

[Buschmann 07] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, Wiley, 2007.*

[Buschmann 07] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages,* Wiley, 2007.

[Avgeriou 05] Paris Avgeriou,Uwe Zdun, "Architectural patterns revisited:a pattern language," 10th European Conference on Pattern Languages of Programs (EuroPlop 2005).

2. Various Architecture Patterns

Acknowledgement

[Gomaa 11] H. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*, Cambridge University Press, 2011.

Table A.1. Software architectural structure patterns

Software architectural structure patterns	Pattern description	Reference chapter
Broker Pattern	Section A.1.1	Chapter 16, Section 16.2
Centralized Control Pattern	Section A.1.2	Chapter 18, Section 18.3.1
Distributed Control Pattern	Section A.1.3	Chapter 18, Section 18.3.2
Hierarchical Control Pattern	Section A.1.4	Chapter 18, Section 18.3.3
Layers of Abstraction Pattern	Section A.1.5	Chapter 12, Section 12.3.1
Multiple Client/Multiple Service Pattern	Section A.1.6	Chapter 15, Section 15.2.2
Multiple Client/Single Service Pattern	Section A.1.7	Chapter 15, Section 15.2.1
Multi-tier Client/Service Pattern	Section A.1.8	Chapter 15, Section 15.2.3

Table A.2. Software architectural communication patterns

Software architectural communication patterns	Pattern description	Reference chapter
Asynchronous Message Communication Pattern	Section A.2.1	Chapter 12, Section 12.3.3
Asynchronous Message Communication with Callback Pattern	Section A.2.2	Chapter 15, Section 15.3.2
Bidirectional Asynchronous Message Communication	Section A.2.3	Chapter 12, Section 12.3.3
Broadcast Pattern	Section A.2.4	Chapter 17, Section 17.6.1
Broker Forwarding Pattern	Section A.2.5	Chapter 16, Section 16.2.2
Broker Handle Pattern	Section A.2.6	Chapter 16, Section 16.2.3
Call/Return	Section A.2.7	Chapter 12, Section 12.3.2
Negotiation Pattern	Section A.2.8	Chapter 16, Section 16.5
Service Discovery Pattern	Section A.2.9	Chapter 16, Section 16.2.4
Service Registration	Section A.2.10	Chapter 16, Section 16.2.1
Subscription/Notification Pattern	Section A.2.11	Chapter 17, Section 17.6.2
Synchronous Message Communication with Reply Pattern	Section A.2.12	Chapter 12, Section 12.3.4; Chapter 15, Section 15.3.1
Synchronous Message Communication without Reply Pattern	Section A.2.13	Chapter 18, Section 18.8.3

Table A.3. Software architectural transaction patterns

Software architectural transaction patterns	Pattern description	Reference chapter
Compound Transaction Pattern	Section A.3.1	Chapter 16, Section 16.4.2
Long-Living Transaction Pattern	Section A.3.2	Chapter 16, Section 16.4.3
Two-Phase Commit Protocol Pattern	Section A.3.3	Chapter 16, Section 16.4.1

A.1.1 Broker Pattern

Pattern name	Broker
Aliases	Object Broker, Object Request Broker
Context	Software architectural design, distributed systems
Problem	Distributed application in which multiple clients communicate with multiple services. Clients do not know locations of services.
Summary of solution	Services register with broker. Clients send service requests to broker. Broker acts as intermediary between client and service.
Strengths of solution	Location transparency: Services may relocate easily. Clients do not need to know locations of services.
Weaknesses of solution	Additional overhead because broker is involved in message communication. Broker can become a bottleneck if there is a heavy load at the broker. Client may keep outdated service handle instead of discarding.
Applicability	Distributed environments: client/service and distribution applications with multiple services
Related patterns	Broker Forwarding, Broker Handle
Reference	Chapter 16, Section 16.2

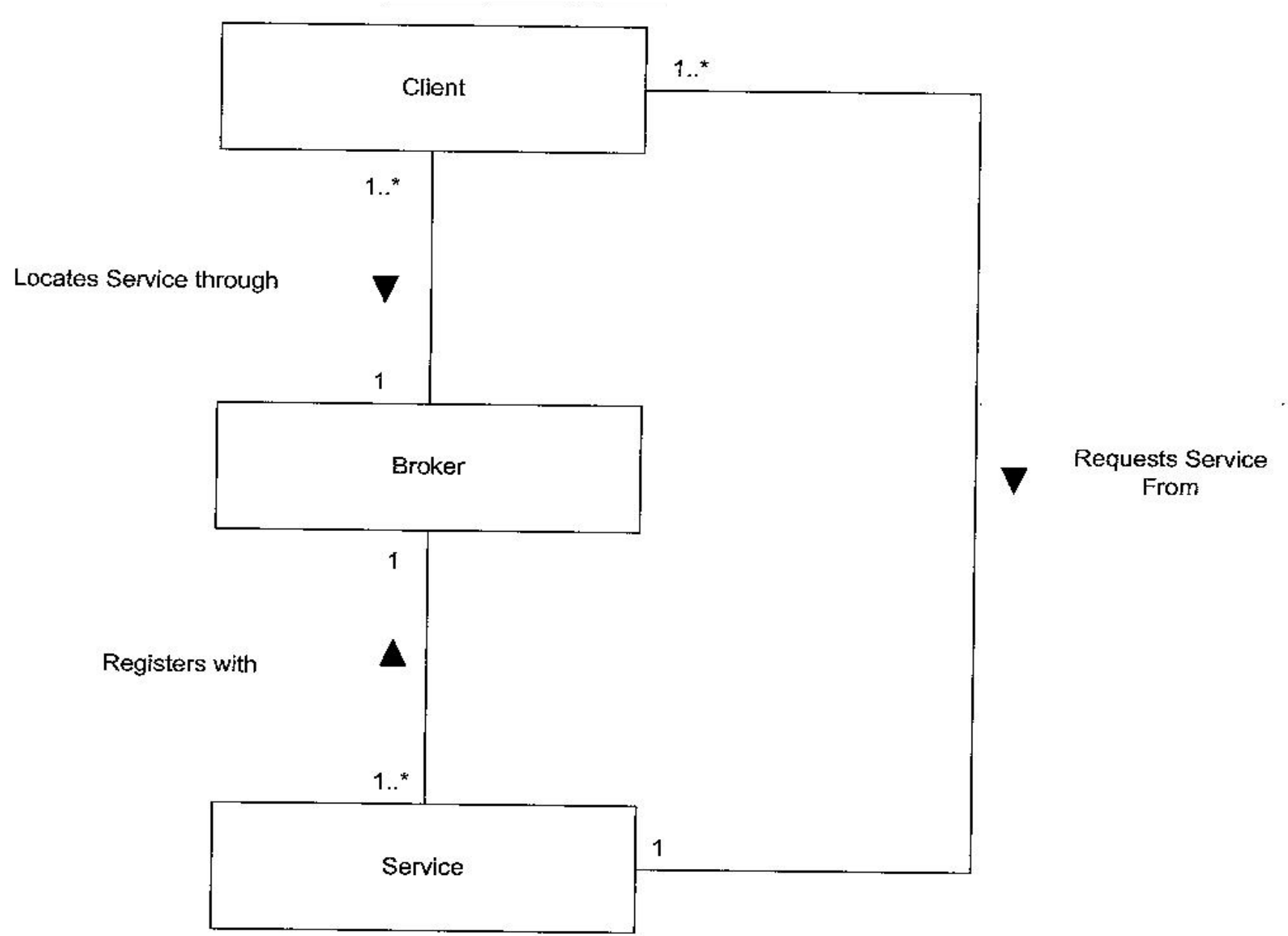


Figure A.1. Broker pattern

A.1.2 Centralized Control Pattern

Pattern name	Centralized Control
Aliases	Centralized Controller, System Controller
Context	Centralized application where overall control is needed
Problem	Several actions and activities are state-dependent and need to be controlled and sequenced.
Summary of solution	There is one control component, which conceptually executes a statechart and provides the overall control and sequencing of the system or subsystem.
Strengths of solution	Encapsulates all state-dependent control in one component
Weaknesses of solution	Could lead to overcentralized control, in which case decentralized control should be considered.
Applicability	Real-time control systems, state-dependent applications
Related patterns	Distributed Control, Hierarchical Control
Reference	Chapter 18, Section 18.3.1

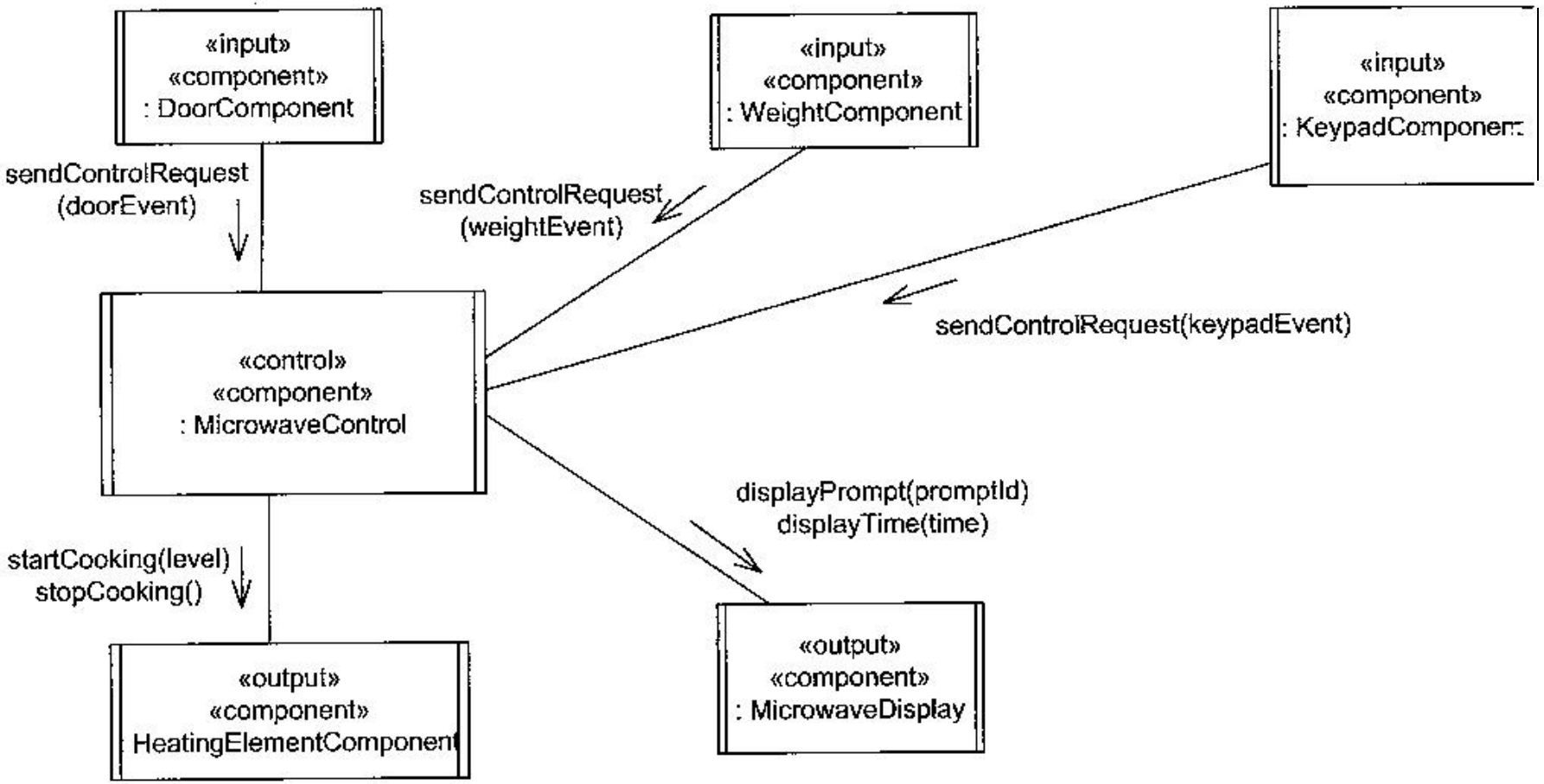


Figure A.2. Centralized Control pattern: Microwave Oven Control System example

A.1.3 Distributed Control Pattern

Pattern name	Distributed Control
Aliases	Distributed Controller
Context	Distributed application with real-time control requirement
Problem	Distributed application with multiple locations where real-time localized control is needed at several locations
Summary of solution	There are several control components, such that each component controls a given part of the system by conceptually executing a state machine. Control is distributed among the various control components; no single component has overall control.
Strengths of solution	Overcomes potential problem of overcentralized control.
Weaknesses of solution	Does not have an overall coordinator. If this is needed, consider using Hierarchical Control pattern.
Applicability	Distributed real-time control, distributed state-dependent applications
Related patterns	Hierarchical Control, Centralized Control
Reference	Chapter 18, Section 18.3.2

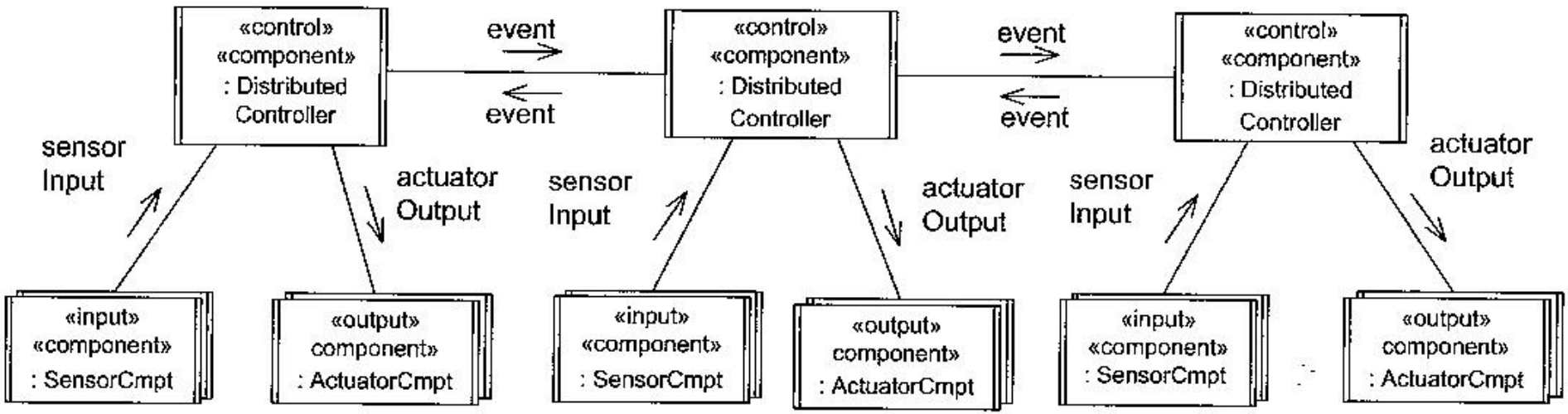


Figure A.3. Distributed Control pattern

A.1.4 Hierarchical Control Pattern

Pattern name	Hierarchical Control
Aliases	Multilevel Control
Context	Distributed application with real-time control requirement
Problem	Distributed application with multiple locations where both real-time localized control and overall control are needed
Summary of solution	There are several control components, each controlling a given part of a system by conceptually executing a statechart. There is also a coordinator component, which provides high-level control by deciding the next job for each control component and communicating that information directly to the control component.
Strengths of solution	Overcomes potential problem with Distributed Control pattern by providing high-level control and coordination
Weaknesses of solution	High-level coordinator may become a bottleneck when the load is high and is a single point of failure.
Applicability	Distributed real-time control, distributed state-dependent applications
Related patterns	Distributed Control, Centralized Control
Reference	Chapter 18, Section 18.3.3

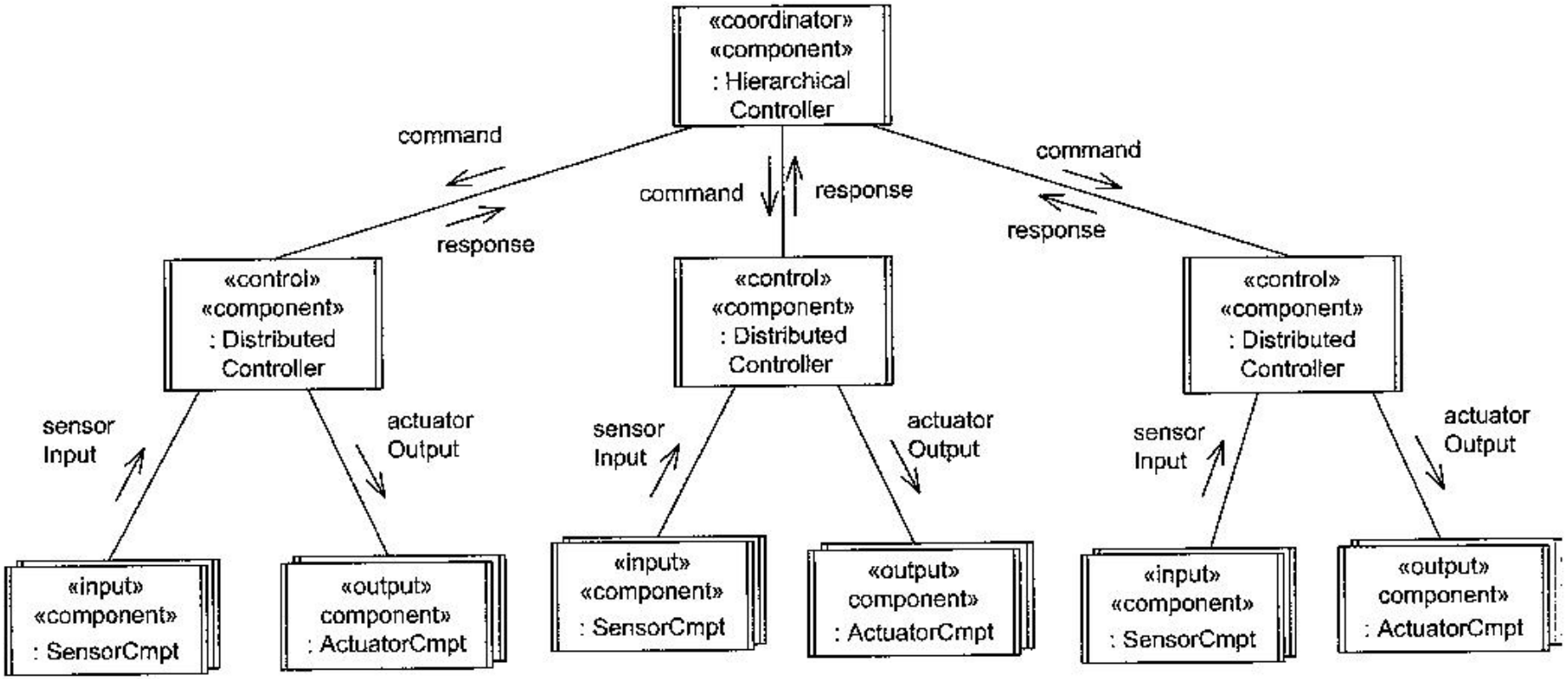


Figure A.4. Hierarchical Control pattern

A.1.5 Layers of Abstraction Pattern

Pattern name	Layers of Abstraction
Aliases	Hierarchical Layers, Levels of Abstraction
Context	Software architectural design
Problem	A software architecture that encourages design for ease of extension and contraction is needed.
Summary of solution	Components at lower layers provide services for components at higher layers. Components may use only services provided by components at lower layers.
Strengths of solution	Promotes extension and contraction of software design
Weaknesses of solution	Could lead to inefficiency if too many layers need to be traversed
Applicability	Operating systems, communication protocols, software product lines
Related patterns	Software kernel can be lowest layer of Layers of Abstraction architecture. Variations of this pattern include Flexible Layers of Abstraction.
Reference	Chapter 12, Section 12. 3.1; Hoffman and Weiss 2001; Parnas 1979

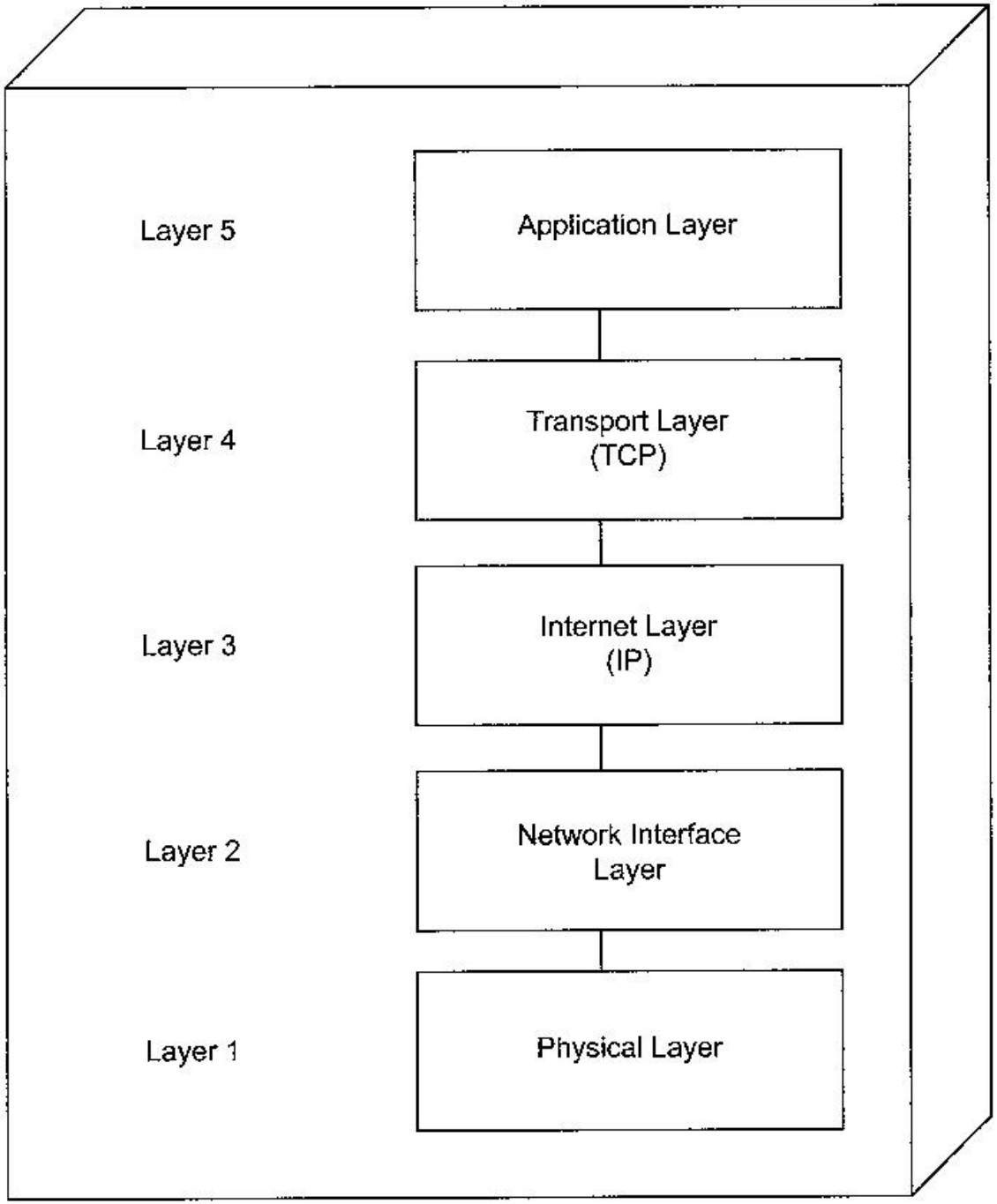


Figure A.5. Layers of Abstraction pattern: TCP/IP example

A.1.6 Multiple Client/Multiple Service Pattern

Pattern name	Multiple Client/Multiple Service
Aliases	Client/Service, Client/Server
Context	Software architectural design, distributed systems
Problem	Distributed application in which multiple clients require services from multiple services
Summary of solution	Client communicates with multiple services, usually sequentially but could also be in parallel. Each service responds to client requests. Each service handles multiple client requests. A service may delegate a client request to a different service.
Strengths of solution	Good way for client to communicate with multiple services when it needs different information from each service.
Weaknesses of solution	Client can be held up indefinitely if there is a heavy load at any server.
Applicability	Distributed processing: client/service and distribution applications with multiple services
Related patterns	Multiple Client/Single Service and Multi-tier Client/Service
Reference	Chapter 15, Section 15.2.2

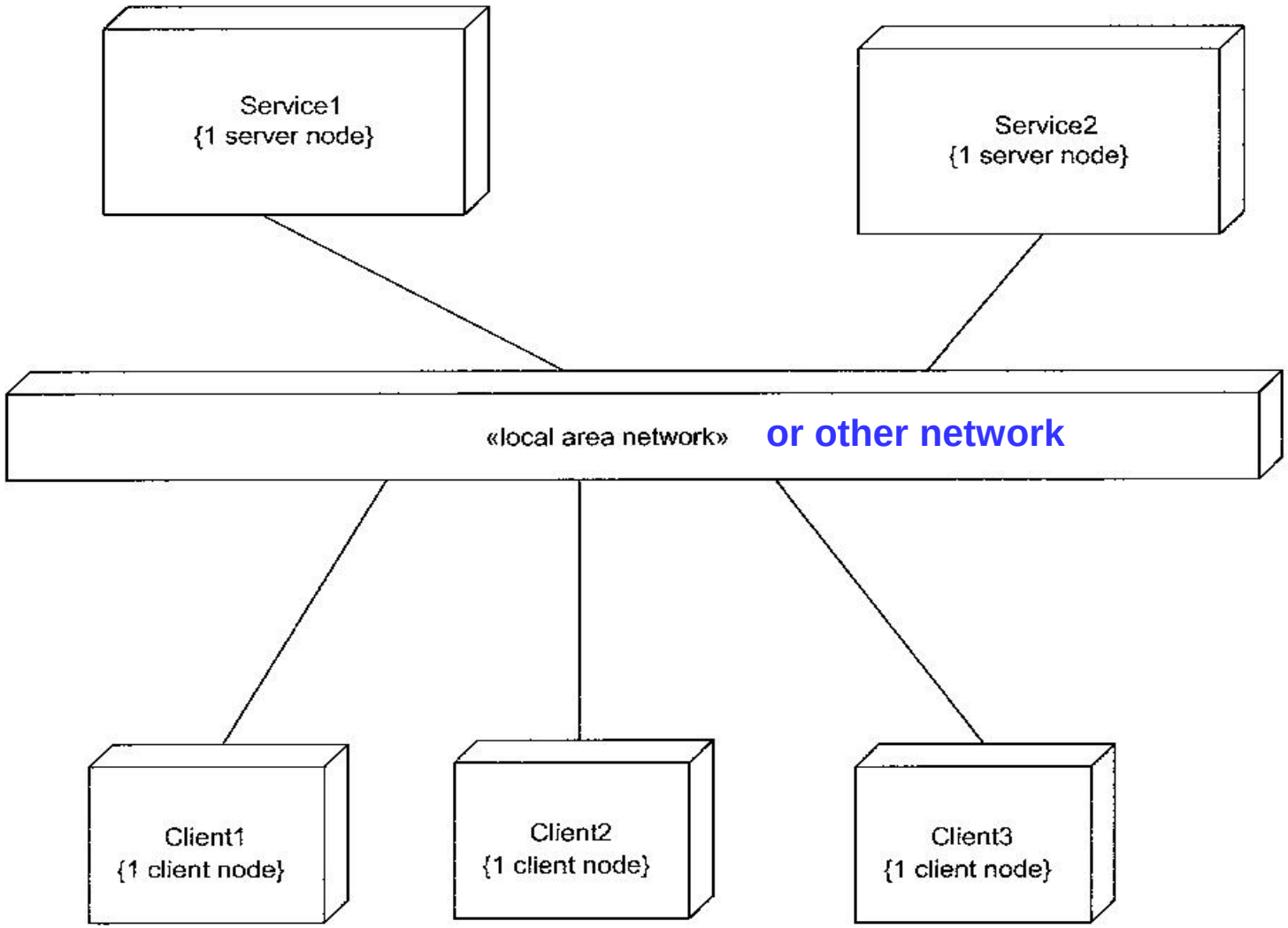


Figure A.6. Multiple Client/Multiple Service Pattern

A.1.7 Multiple Client/Single Service Pattern

Pattern name	Multiple Client/Single Service
Aliases	Client/Service, Client/Server
Context	Software architectural design, distributed systems
Problem	Distributed application in which multiple clients require services from a single service
Summary of solution	Client requests service. Service responds to client requests and does not initiate requests. Service handles multiple client requests.
Strengths of solution	Good way for client to communicate with service when it needs a reply from service. Very common form of communication in client/server applications.
Weaknesses of solution	Client can be held up indefinitely if there is a heavy load at the server.
Applicability	Distributed processing: client/service applications
Related patterns	Multiple Client/Multiple Service and Multi-tier Client/Service
Reference	Chapter 15, Section 15.2.1

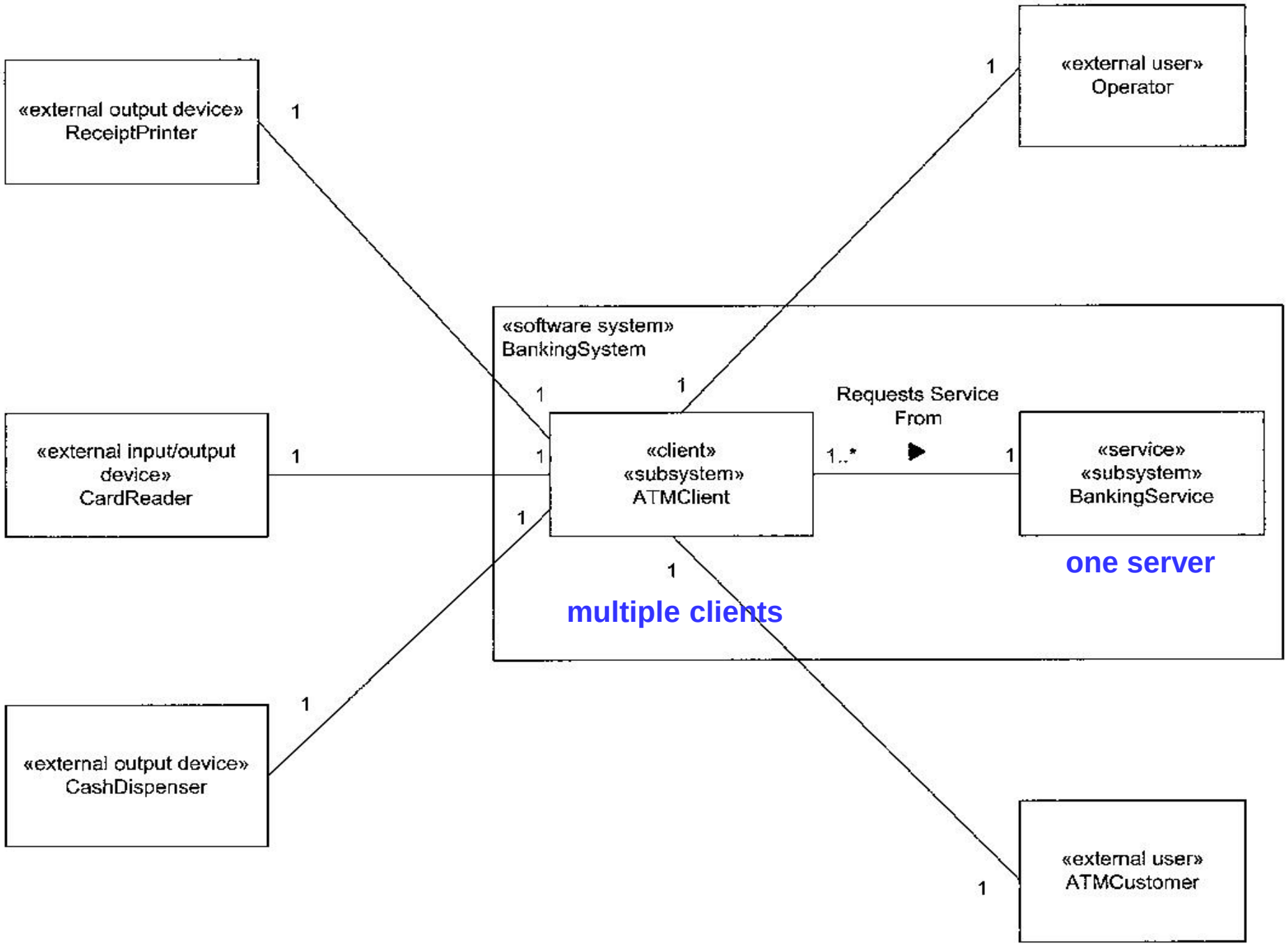


Figure A.7. Multiple Client/Single Service Pattern: Banking System example

A.1.8 Multi-tier Client/Service Pattern

Pattern name	Multi-tier Client/Service
Aliases	Client/Service, Client/Server
Context	Software architectural design, distributed systems
Problem	Distributed application in which there is more than one tier (layer) of service
Summary of solution	Client requests service. Solution consists of more than one tier of service. Intermediate tier provides both client and service role. There can be more than one intermediate tier.
Strengths of solution	Good way of layering services if multiple services are needed to handle an individual client's request and one service needs assistance of another service.
Weaknesses of solution	Client can be held up indefinitely if there is a heavy load at the server.
Applicability	Distributed processing: client/service and distribution applications with multiple services
Related patterns	Multiple Client/Single Service and Multiple Client/Multiple Service
Reference	Chapter 15, Section 15.2.3

Figure A.8. Multi-tier Client/Service Pattern: Banking System example

A.2.1 Asynchronous Message Communication Pattern

Pattern name	Asynchronous Message Communication
Aliases	Loosely Coupled Message Communication
Context	Concurrent or distributed systems
Problem	Concurrent or distributed application has concurrent components that need to communicate with each other. Producer does not need to wait for consumer. Producer does not need a reply.
Summary of solution	Use message queue between producer component and consumer component. Producer sends message to consumer and continues. Consumer receives message. Messages are queued FIFO if consumer is busy. Consumer is suspended if no message is available. Producer needs timeout notification if consumer node is down.
Strengths of solution	Consumer does not hold up producer.
Weaknesses of solution	If producer produces messages more quickly than consumer can process them, the message queue will eventually overflow.
Applicability	Centralized and distributed environments: real-time systems, client/server and distribution applications
Related patterns	Asynchronous Message Communication with Callback
Reference	Chapter 12, Section 12.3.3

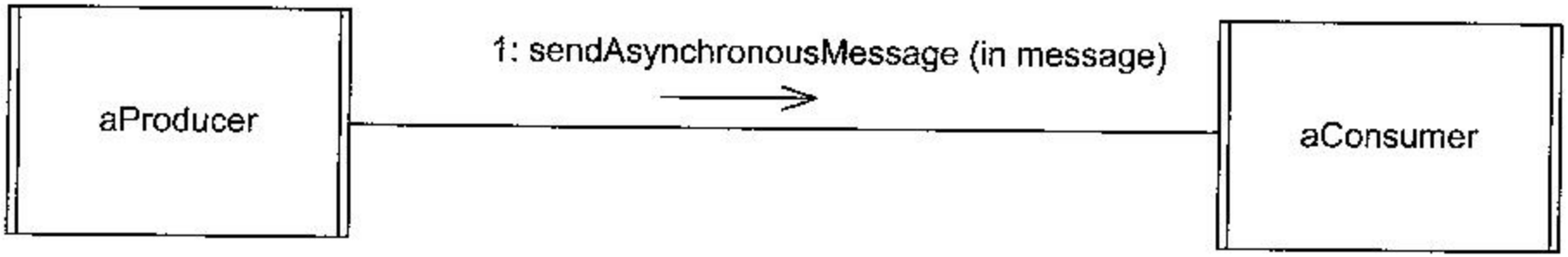


Figure A.9. Asynchronous Message Communication pattern

A.2.2 Asynchronous Message Communication with Callback Pattern

Pattern name	Asynchronous Message Communication with Callback
Aliases	Loosely Coupled Communication with Callback
Context	Concurrent or distributed systems
Problem	Concurrent or distributed application in which concurrent components need to communicate with each other. Client does not need to wait for service but does need to receive a reply later.
Summary of solution	Use synchronous communication between clients and service. Client sends request to service, which includes client operation (callback) handle. Client does not wait for reply. After service processes the client request, it uses the handle to call the client operation remotely (the callback).
Strengths of solution	Good way for client to communicate with service when it needs a reply but can continue executing and receive reply later
Weaknesses of solution	Suitable only if the client does not need to send multiple requests before receiving the first reply
Applicability	Distributed environments: client/server and distribution applications with multiple servers
Related patterns	Consider Bidirectional Asynchronous Message Communication as alternative pattern.
Reference	Chapter 15, Section 15.3.2

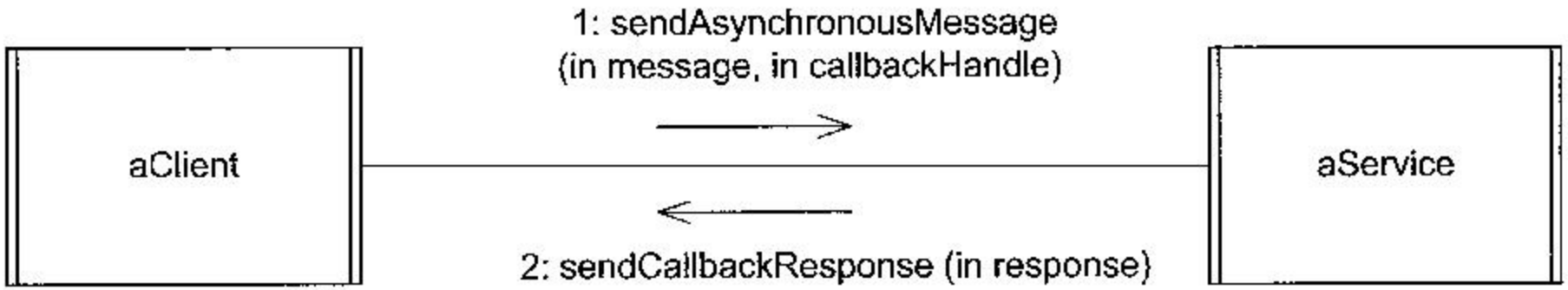


Figure A.10. Asynchronous Message Communication with Callback pattern

A.2.3 Bidirectional Asynchronous Message Communication Pattern

Pattern name	Bidirectional Asynchronous Message Communication
Aliases	Bidirectional Loosely Coupled Message Communication
Context	Concurrent or distributed systems
Problem	Concurrent or distributed application in which concurrent components need to communicate with each other. Producer does not need to wait for consumer, although it does need to receive replies later. Producer can send several requests before receiving first reply.
Summary of solution	Use two message queues between producer component and consumer component: one for messages from producer to consumer, and one for messages from consumer to producer. Producer sends message to consumer on P→C queue and continues. Consumer receives message. Messages are queued if consumer is busy. Consumer sends replies on C→P queue.
Strengths of solution	Producer does not get held up by consumer. Producer receives replies later, when it needs them.
Weaknesses of solution	If producer produces messages more quickly than consumer can process them, the message (P→C) queue will eventually overflow. If producer does not service replies quickly enough, the reply (C→P) queue will overflow.
Applicability	Centralized and distributed environments: real-time systems, client/server and distribution applications
Related patterns	Asynchronous Message Communication with Callback
Reference	Chapter 12, Section 12.3.3

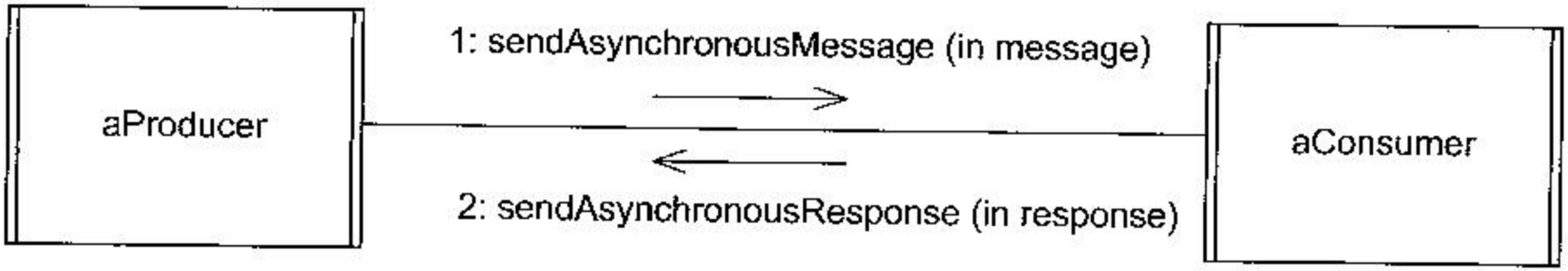


Figure A.11. Bidirectional Asynchronous Message Communication pattern

A.2.4 Broadcast Pattern

Pattern name	Broadcast
Aliases	Broadcast Communication
Context	Distributed systems
Problem	Distributed application with multiple clients and services. At times, a service needs to send the same message to several clients.
Summary of solution	Crude form of group communication in which service sends a message to all clients, regardless of whether clients want the message or not. Client decides whether it wants to process the message or just discard the message.
Strengths of solution	Simple form of group communication
Weaknesses of solution	Places an additional load on the client, because the client may not want the message
Applicability	Distributed environments: client/server and distribution applications with multiple servers
Related patterns	Similar to Subscription/Notification, except that it is not selective
Reference	Chapter 17, Section 17.6.1

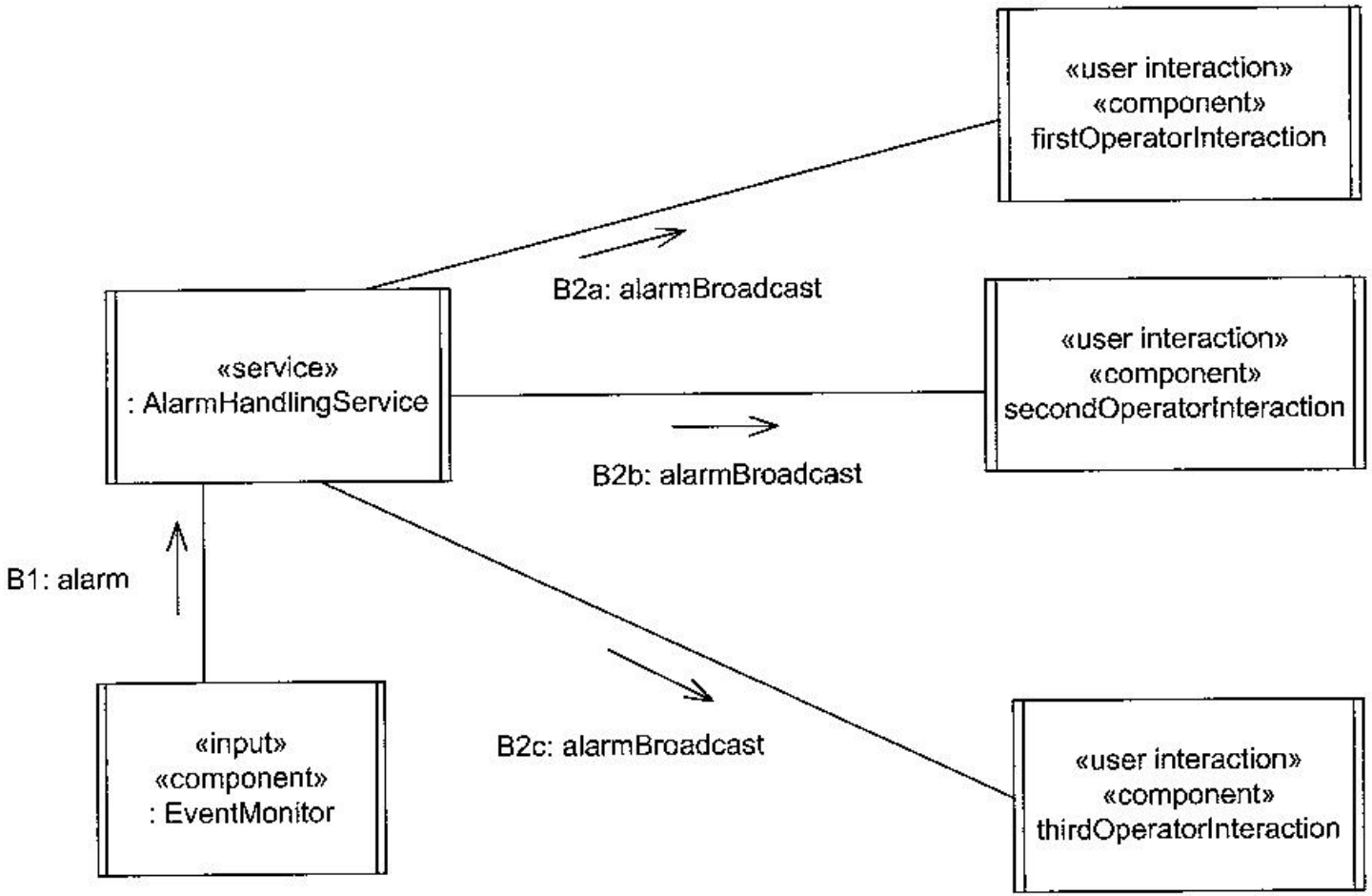


Figure A.12. Broadcast pattern: alarm broadcast example

A.2.5 Broker Forwarding Pattern

Pattern name	Broker Forwarding
Aliases	White Pages Broker Forwarding, Broker with Forwarding Design
Context	Distributed systems
Problem	Distributed application in which multiple clients communicate with multiple services. Clients do not know locations of services.
Summary of solution	Services register with broker. Client sends service request to broker. Broker forwards request to service. Service processes request and sends reply to broker. Broker forwards reply to client.
Strengths of solution	Location transparency: Services may relocate easily. Clients do not need to know locations of services.
Weaknesses of solution	Additional overhead because broker is involved in all message communication. Broker can become a bottleneck if there is a heavy load at the broker.
Applicability	Distributed environments: client/server and distribution applications with multiple servers
Related patterns	Similar to Broker Handle; more secure, but performance is not as good
Reference	Chapter 16, Section 16.2.2

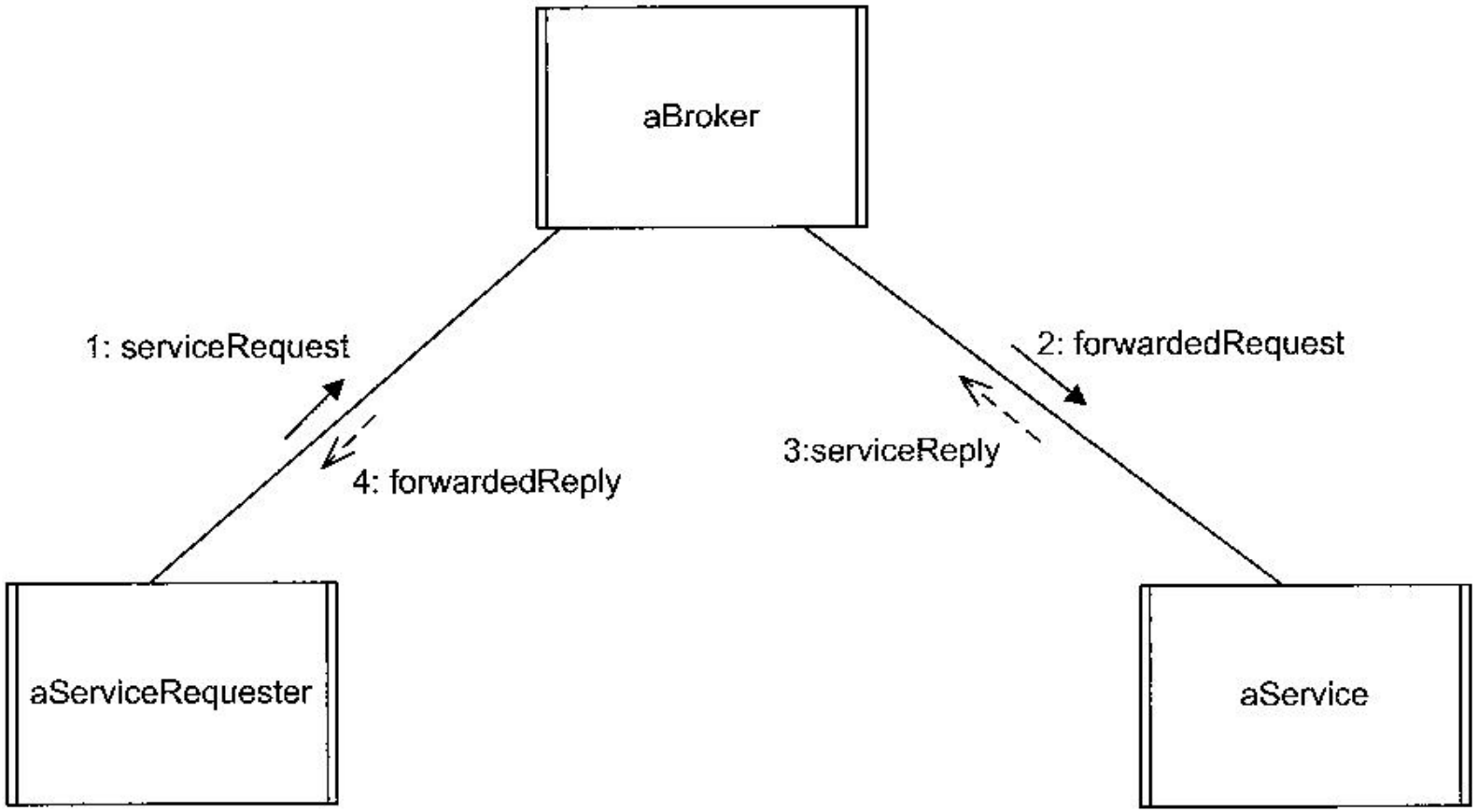


Figure A.13. Broker Forwarding pattern

A.2.6 Broker Handle Pattern

Pattern name	Broker Handle
Aliases	White Pages Broker Handle, Broker with Handle-Driven Design
Context	Distributed systems
Problem	Distributed application in which multiple clients communicate with multiple services. Clients do not know locations of services.
Summary of solution	Services register with broker. Client sends service request to broker. Broker returns service handle to client. Client uses service handle to make request to service. Service processes request and sends reply directly to client. Client can make multiple requests to service without broker involvement.
Strengths of solution	Location transparency: Services may relocate easily. Clients do not need to know locations of services.
Weaknesses of solution	Additional overhead because broker is involved in initial message communication. Broker can become a bottleneck if there is a heavy load at the broker. Client may keep outdated service handle instead of discarding.
Applicability	Distributed environments: client/server and distribution applications with multiple servers
Related patterns	Similar to Broker Forwarding, but with better performance
Reference	Chapter 16, Section 16.2.3

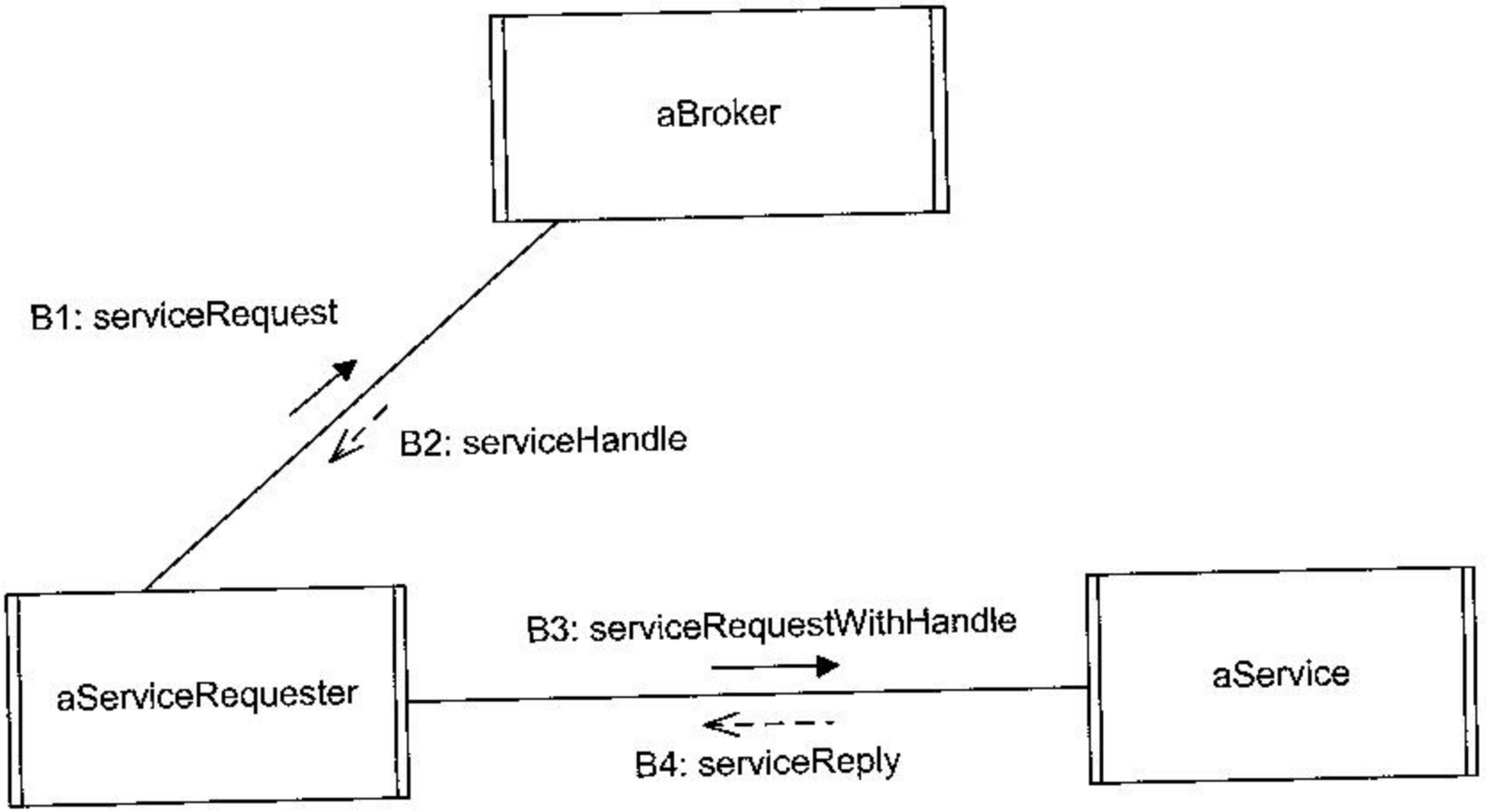


Figure A.14. Broker Handle pattern

A.2.7 Call/Return Pattern

Pattern name	Call/Return
Aliases	Operation invocation, method invocation
Context	Object-oriented programs and systems
Problem	An object needs to call an operation (also known as method) in a different object.
Summary of solution	A calling operation in a calling object invokes a called operation in a called object. Control is passed, together with any input parameters, from the calling operation to the called operation at the time of operation invocation. When the called operation finishes executing, it returns control and any output parameters to the calling operation.
Strengths of solution	This pattern is the only possible form of communication between objects in a sequential design.
Weaknesses of solution	If this pattern of communication is not suitable, then most likely a concurrent or distributed solution will be needed.
Applicability	Sequential object-oriented architectures, programs, and systems. A service designed as a sequential subsystem that communicates with internal objects using this pattern.
Related patterns	Software architectural communication patterns in which message passing is used instead of operation invocation.
Reference	Chapter 12, Section 12.3.2

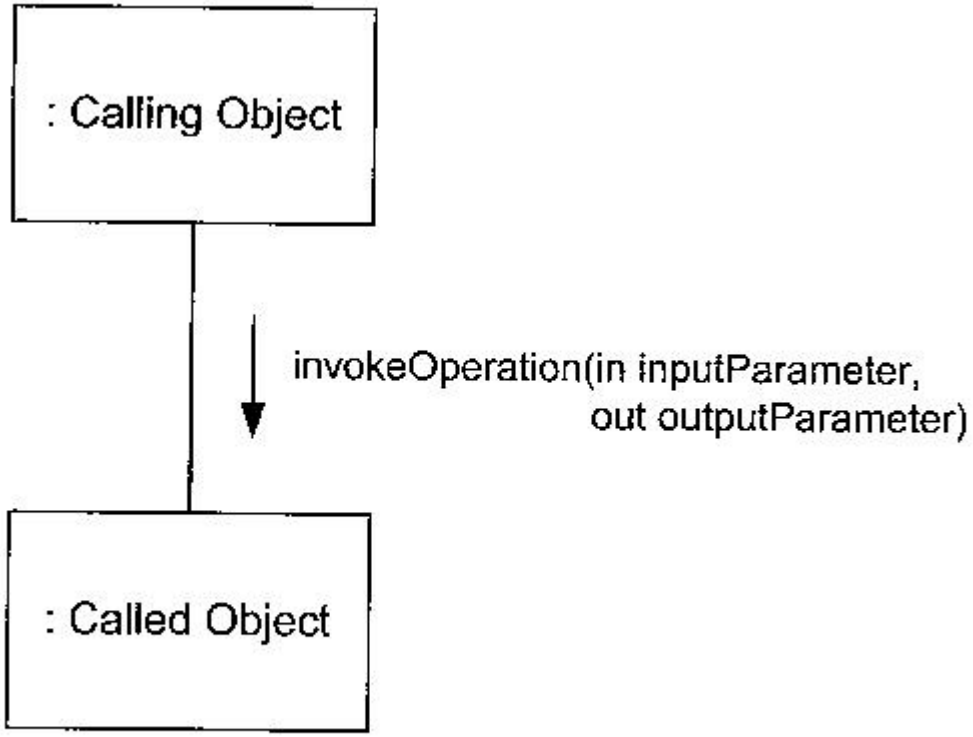


Figure A.15. Call/Return pattern

A.2.8 Negotiation Pattern

Pattern name	Negotiation
Aliases	Agent-Based Negotiation, Multi-Agent Negotiation
Context	Distributed multi-agent systems; service-oriented architectures
Problem	Client needs to negotiate with multiple services to find best available service.
Summary of solution	Client agent acts on behalf of client and makes a proposal to service agent, who acts on behalf of service. Service agent attempts to satisfy client's proposal, which might involve communication with other services. Having determined the available options, service agent then offers client agent one or more options that come closest to matching the original client agent proposal. Client agent may then request one of the options, propose further options, or reject the offer. If service agent can satisfy client agent request, client agent accepts the request; otherwise, it rejects the request.
Strengths of solution	Provides negotiation service to complement other services
Weaknesses of solution	Negotiation may be lengthy and inconclusive.
Applicability	Distributed environments: client/service and distribution applications with multiple services, service-oriented architectures
Related patterns	Often used in conjunction with broker patterns (Broker Forwarding, Broker Handle, Service Discovery)
Reference	Chapter 16, Section 16.5

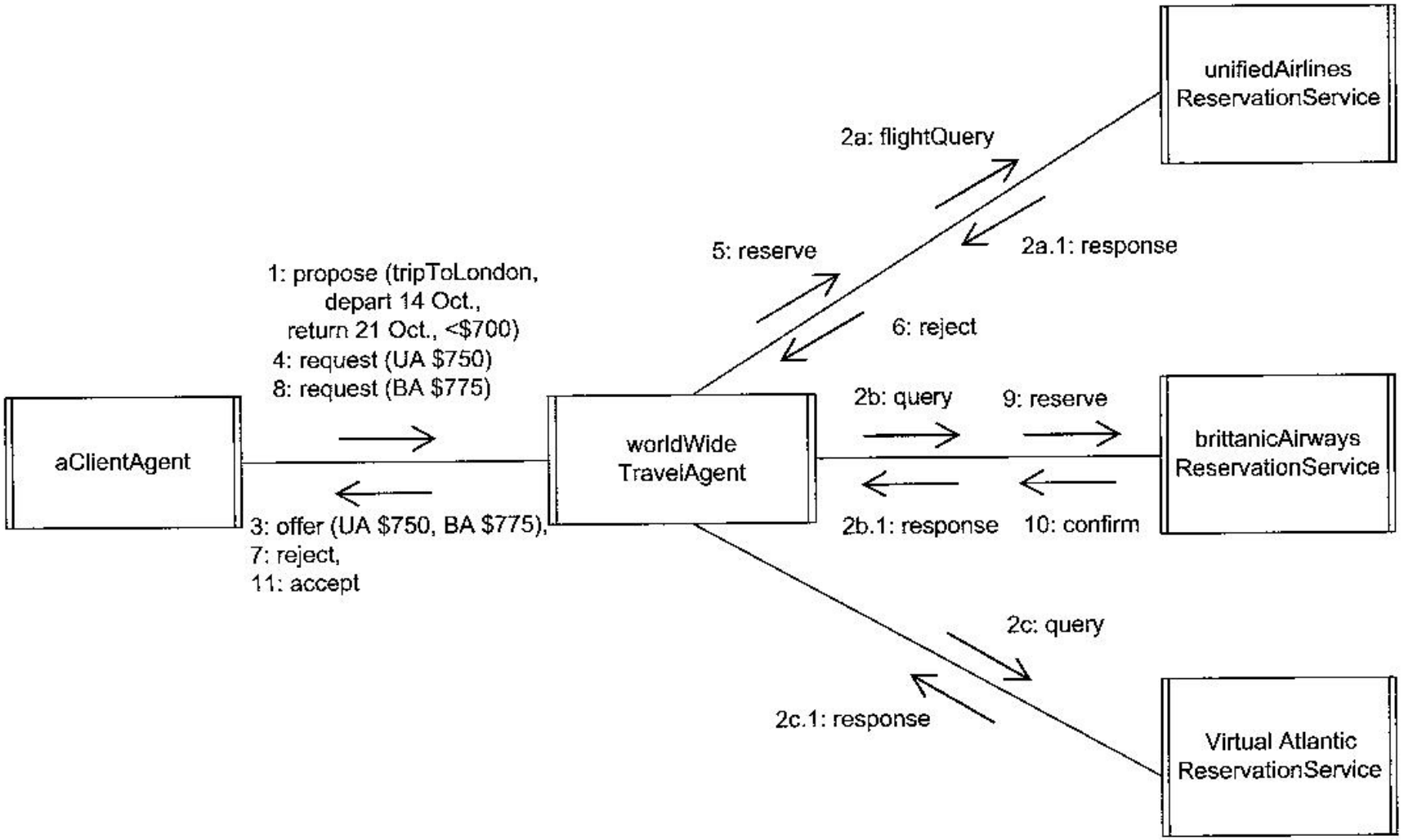


Figure A.16. Negotiation pattern: airline reservation example

A.2.9 Service Discovery Pattern

Pattern name	Service Discovery
Aliases	Yellow Pages Broker, Broker Trader, Discovery
Context	Distributed systems
Problem	Distributed application in which multiple clients communicate with multiple services. Client knows the type of service required but not the specific service.
Summary of solution	Use broker's discovery service. Services register with broker. Client sends discovery service request to broker. Broker returns names of all services that match discovery service request. Client selects a service and uses Broker Handle or Broker Forwarding pattern to communicate with service.
Strengths of solution	Location transparency: Services may relocate easily. Clients do not need to know specific service, only the service type.
Weaknesses of solution	Additional overhead because broker is involved in initial message communication. Broker can become a bottleneck if there is a heavy load at the broker.
Applicability	Distributed environments: client/service and distribution applications with multiple services
Related patterns	Other broker patterns (Broker Forwarding, Broker Handle)
Reference	Chapter 16, Section 16.2.4

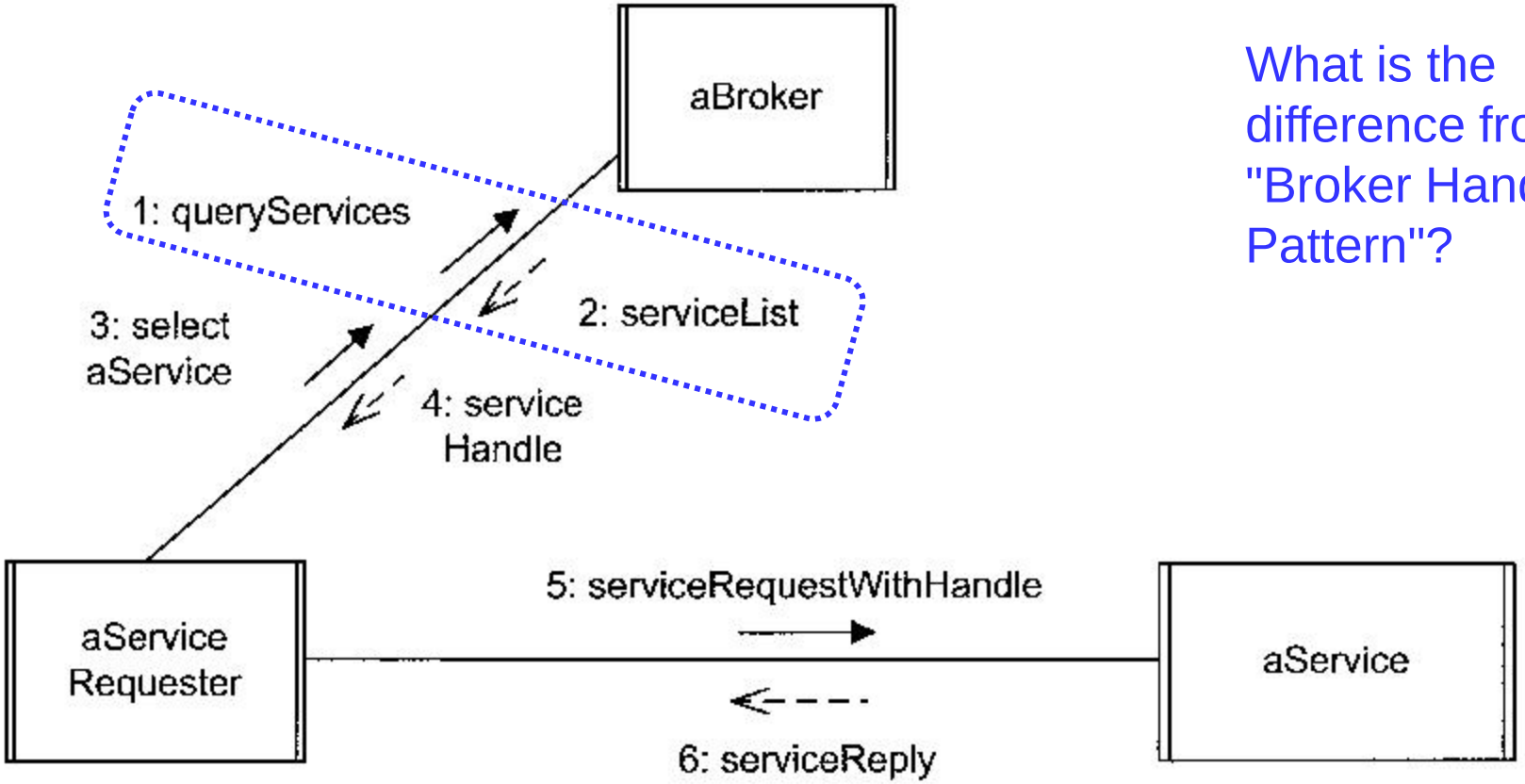


Figure A.17. Service Discovery pattern

What is the difference from "Broker Handle Pattern"?

A.2.10 Service Registration Pattern

Pattern name	Service Registration
Aliases	Broker Registration
Context	Software architectural design, distributed systems
Problem	Distributed application in which multiple clients communicate with multiple services. Clients do not know locations of services.
Summary of solution	Services register service information with broker, including service name, service description, and location. Clients send service requests to broker. Broker acts as intermediary between client and service.
Strengths of solution	Location transparency: Services may relocate easily. Clients do not need to know locations of services.
Weaknesses of solution	Additional overhead because broker is involved in message communication. Broker can become a bottleneck if there is a heavy load at the broker.
Applicability	Distributed environments: client/service and distribution applications with multiple services
Related patterns	Broker, Broker Forwarding, Broker Handle, Service Discover,
Reference	Chapter 16, Section 16.2.1

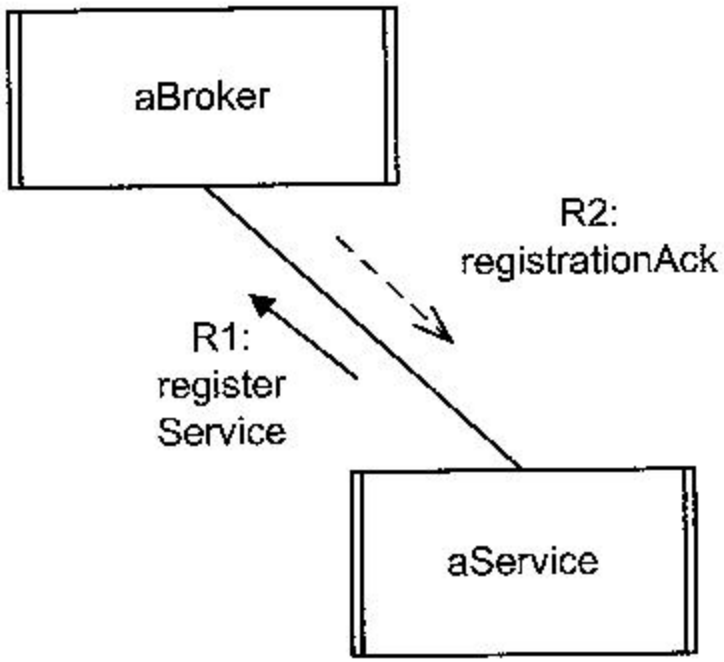


Figure A.18. Service Registration pattern

A.2.11 Subscription/Notification Pattern

Pattern name	Subscription/Notification
Aliases	Multicast
Context	Distributed systems
Problem	Distributed application with multiple clients and services. Clients want to receive messages of a given type.
Summary of solution	Selective form of group communication. Clients subscribe to receive messages of a given type. When service receives message of this type, it notifies all clients who have subscribed to it.
Strengths of solution	Selective form of group communication. Widely used on the Internet and in World Wide Web applications.
Weaknesses of solution	If client subscribes to too many services, it may unexpectedly receive a large number of messages.
Applicability	Distributed environments: client/service and distribution applications with multiple services
Related patterns	Similar to Broadcast, except that it is more selective
Reference	Chapter 17, Section 17.6.2

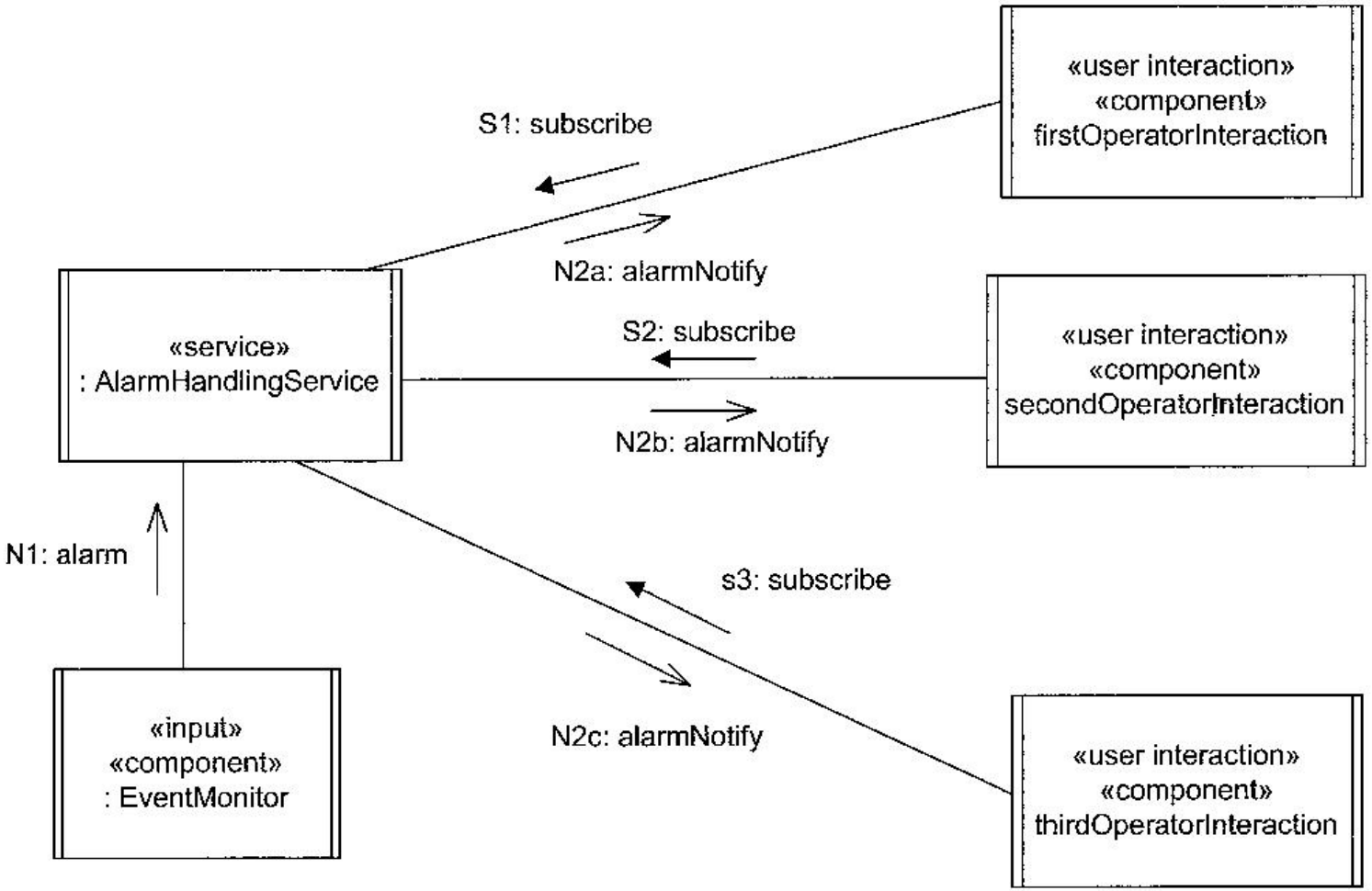


Figure A.19. Subscription/Notification pattern: alarm notification example

A.2.12 Synchronous Message Communication with Reply Pattern

Pattern name	Synchronous Message Communication* with Reply
Aliases	Tightly Coupled Message Communication with Reply
Context	Concurrent or distributed systems
Problem	Concurrent or distributed application in which multiple clients communicate with a single service. Client needs to wait for reply from service.
Summary of solution	Use synchronous communication between clients and service. Client sends message to service and waits for reply. Use message queue at service because there are many clients. Service processes message FIFO. Service sends reply to client. Client is activated when it receives reply from service.
Strengths of solution	Good way for client to communicate with service when it needs a reply. Very common form of communication in client/server applications.
Weaknesses of solution	Client can be held up indefinitely if there is a heavy load at the server.
Applicability	Distributed environments: client/service and distribution applications with multiple services
Related patterns	Asynchronous Message Communication with Callback
Reference	Chapter 12, Section 12.3.4; Chapter 15, Section 15.3.1

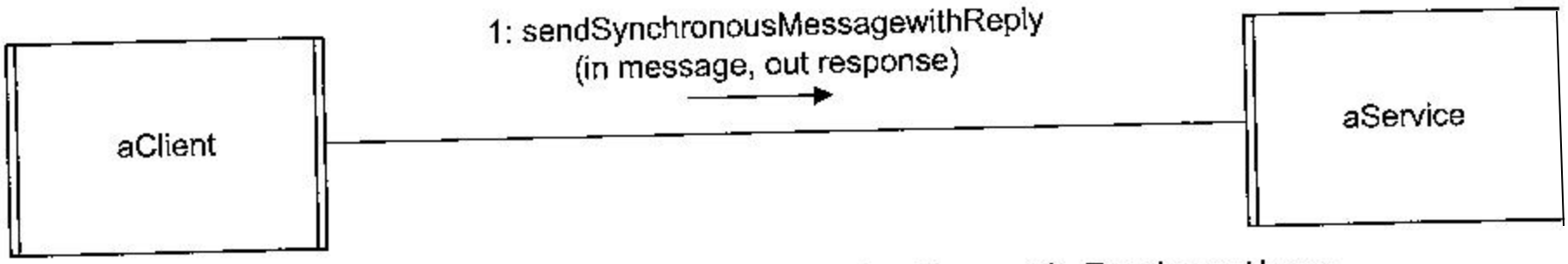


Figure A.20. Synchronous Message Communication with Reply pattern

A.2.13 Synchronous Message Communication without Reply Pattern

Pattern name **Aliases** **Context**	Synchronous Message Communication without Reply Tightly Coupled Message Communication without Reply Concurrent or distributed systems
Problem	Concurrent or distributed application in which concurrent components need to communicate with each other. Producer needs to wait for consumer to accept message. Producer does not want to get ahead of consumer. There is no queue between producer and consumer.
Summary of solution	Use synchronous communication between producer and consumer. Producer sends message to consumer and waits for consumer to accept message. Consumer receives message. Consumer is suspended if no message is available. Consumer accepts message, thereby releasing producer.
Strengths of solution	Good way for producer to communicate with consumer when it wants confirmation that consumer received the message and producer does not want to get ahead of consumer.
Weaknesses of solution	Producer can be held up indefinitely if consumer is busy doing something else.
Applicability	Distributed environments: client/service and distribution applications with multiple services
Related patterns	Consider Synchronous Message Communication with Reply as alternative pattern.
Reference	Chapter 18, Section 18.8.3

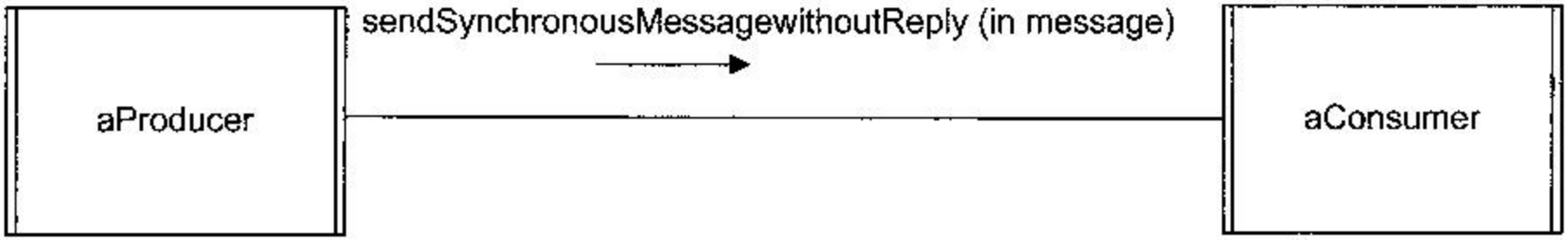


Figure A.21. Synchronous Message Communication without Reply pattern

A.3.1 Compound Transaction Pattern

Pattern name	Compound Transaction
Aliases	
Context	Distributed systems, distributed databases
Problem	Client has a transaction requirement that can be broken down into smaller, separate flat transactions.
Summary of solution	Break down compound transaction into smaller atomic transactions, where each atomic transaction can be performed separately and rolled back separately.
Strengths of solution	Provides effective support for transactions that can be broken into two or more atomic transactions. Effective if a rollback or change is required to only one of the transactions.
Weaknesses of solution	More work is required to make sure that the individual atomic transactions are consistent with each other. More coordination is required if the whole compound transaction needs to be rolled back or modified.
Applicability	Transaction processing applications, distributed databases
Related patterns	Two-Phase Commit Protocol, Long-Living Transaction
Reference	Chapter 16, Section 16.4.2

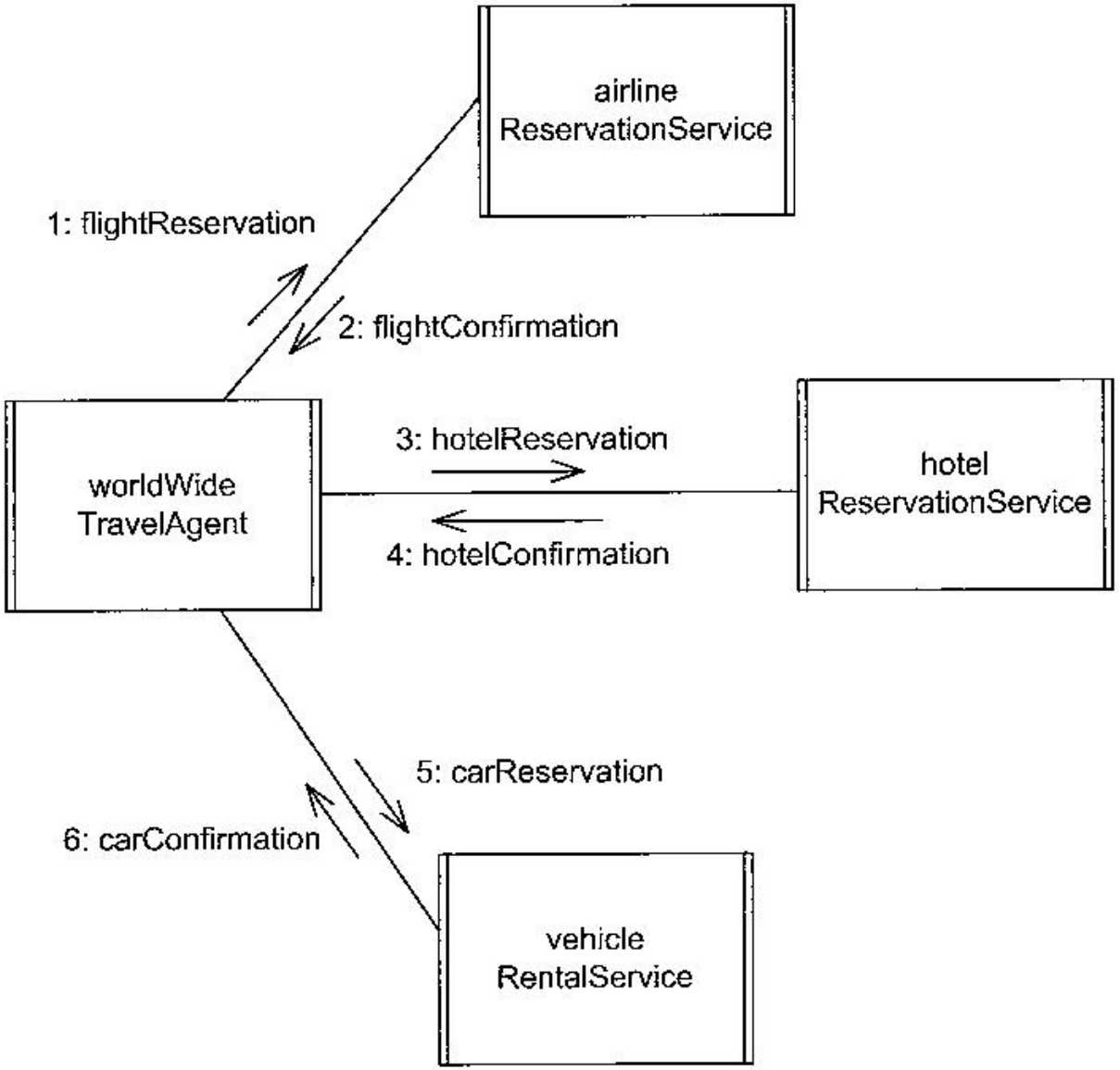


Figure A.22. Compound Transaction pattern: airline/hotel/car reservation example

A.3.2 Long-Living Transaction Pattern

Pattern name	Long-Living Transaction
Aliases	
Context	Distributed systems, distributed databases
Problem	Client has a long-living transaction requirement that has a human in the loop and that could take a long and possibly indefinite time to execute.
Summary of solution	Split a long-living transaction into two or more separate atomic transactions such that human decision making takes place between each successive pair of atomic transactions.
Strengths of solution	Provides effective support for long-living transactions that can be broken into two or more atomic transactions
Weaknesses of solution	Situations may change because of long delay between successive atomic transactions that constitute the long-living transaction, resulting in an unsuccessful long-living transaction.
Applicability	Transaction processing applications, distributed databases
Related patterns	Two-Phase Commit Protocol, Compound Transaction.
Reference	Chapter 16, Section 16.4.3

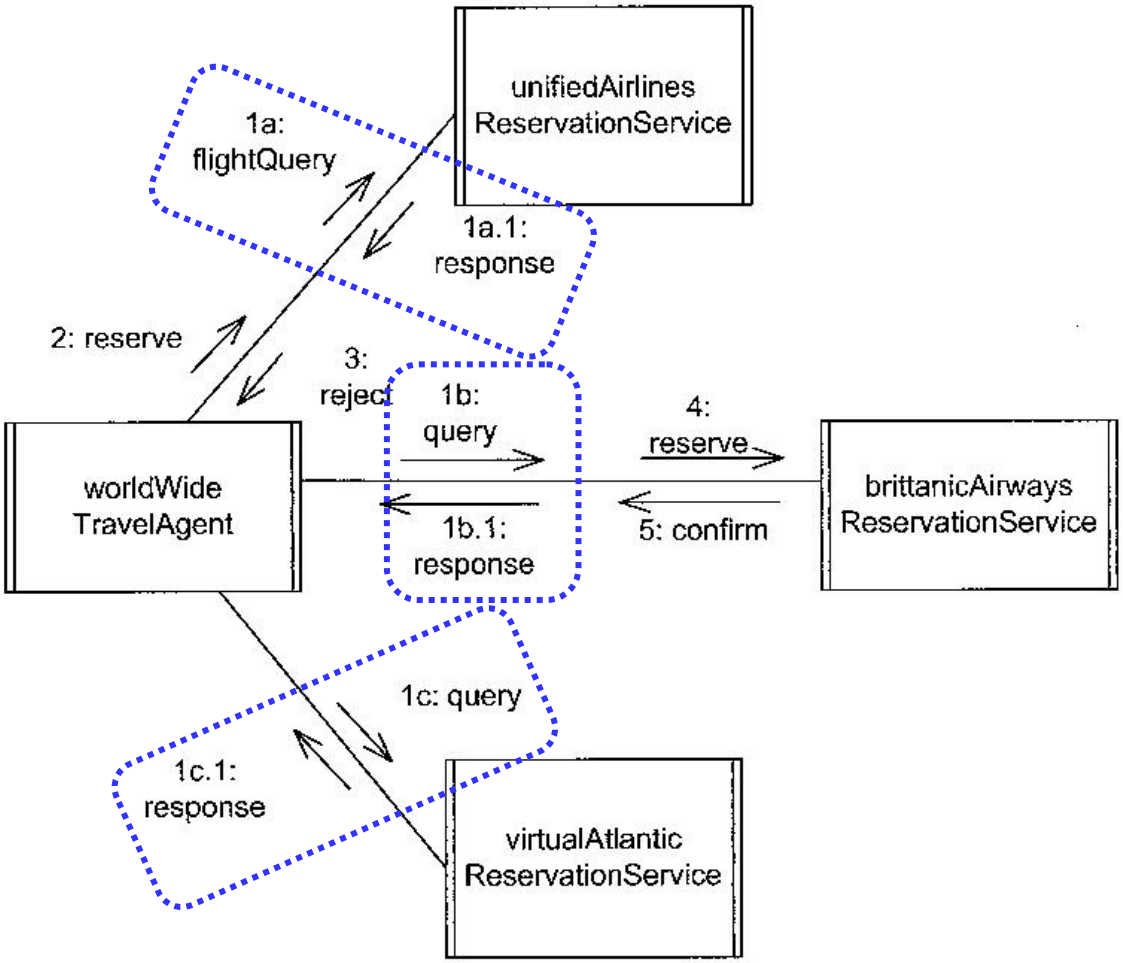
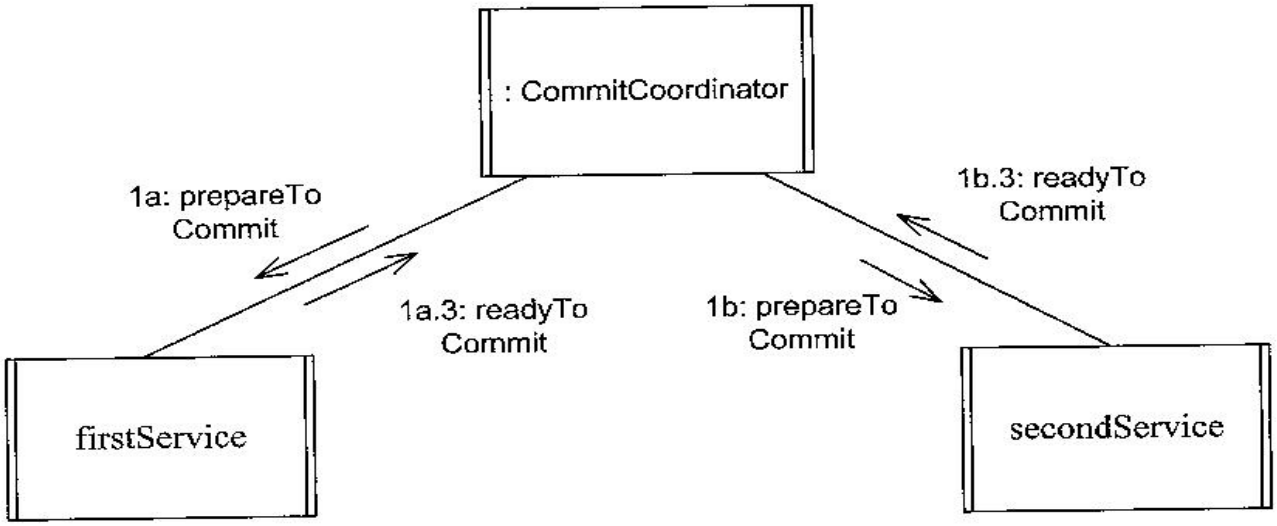


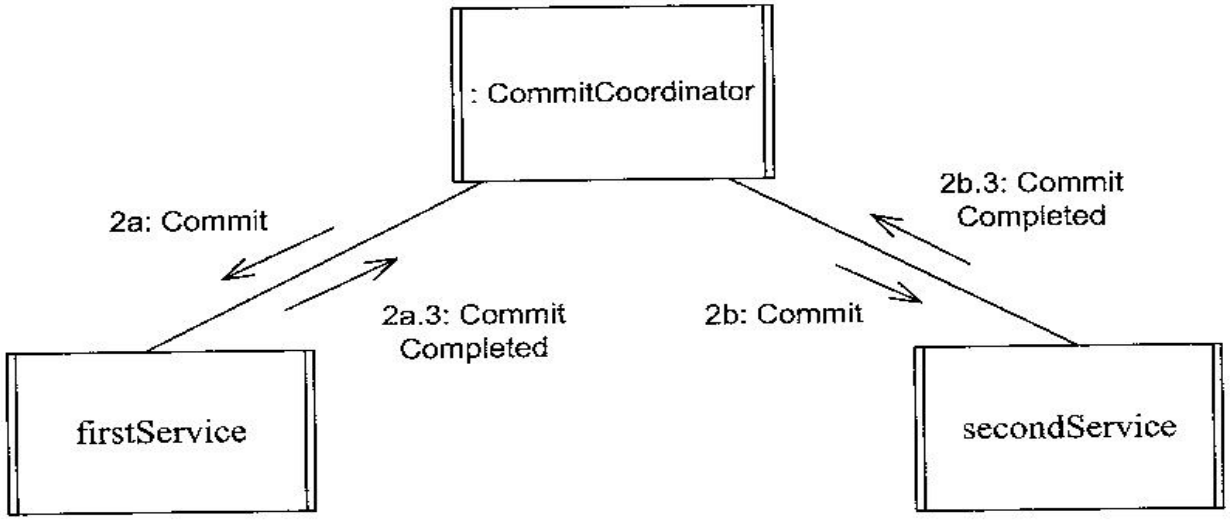
Figure A.23. Long-Living Transaction pattern: airline reservation example

A.3.3 Two-Phase Commit Protocol Pattern

Pattern name	Two-Phase Commit Protocol
Aliases	Atomic Transaction
Context	Distributed systems, distributed databases
Problem	Clients generate transactions and send them to the service for processing. A transaction is atomic (i.e., indivisible). It consists of two or more operations that perform a single logical function, and it must be completed in its entirety or not at all.
Summary of solution	For atomic transactions, services needed to commit or abort the transaction. The two-phase commit protocol is used to synchronize updates on different nodes in distributed applications. The result is that either the transaction is committed (in which case all updates succeed) or the transaction is aborted (in which case all updates fail).
Strengths of solution	Provides effective support for atomic transactions
Weaknesses of solution	Effective only for short transactions; that is, there are no long delays between the two phases of the transaction.
Applicability	Transaction processing applications, distributed databases
Related patterns	Compound Transaction, Long-Living Transaction
Reference	Chapter 16, Section 16.4.1



(a) First phase of Two-Phase Commit Protocol



(b) Second phase of Two-Phase Commit Protocol

Figure A.24. Two-Phase Commit Protocol pattern

Questions?