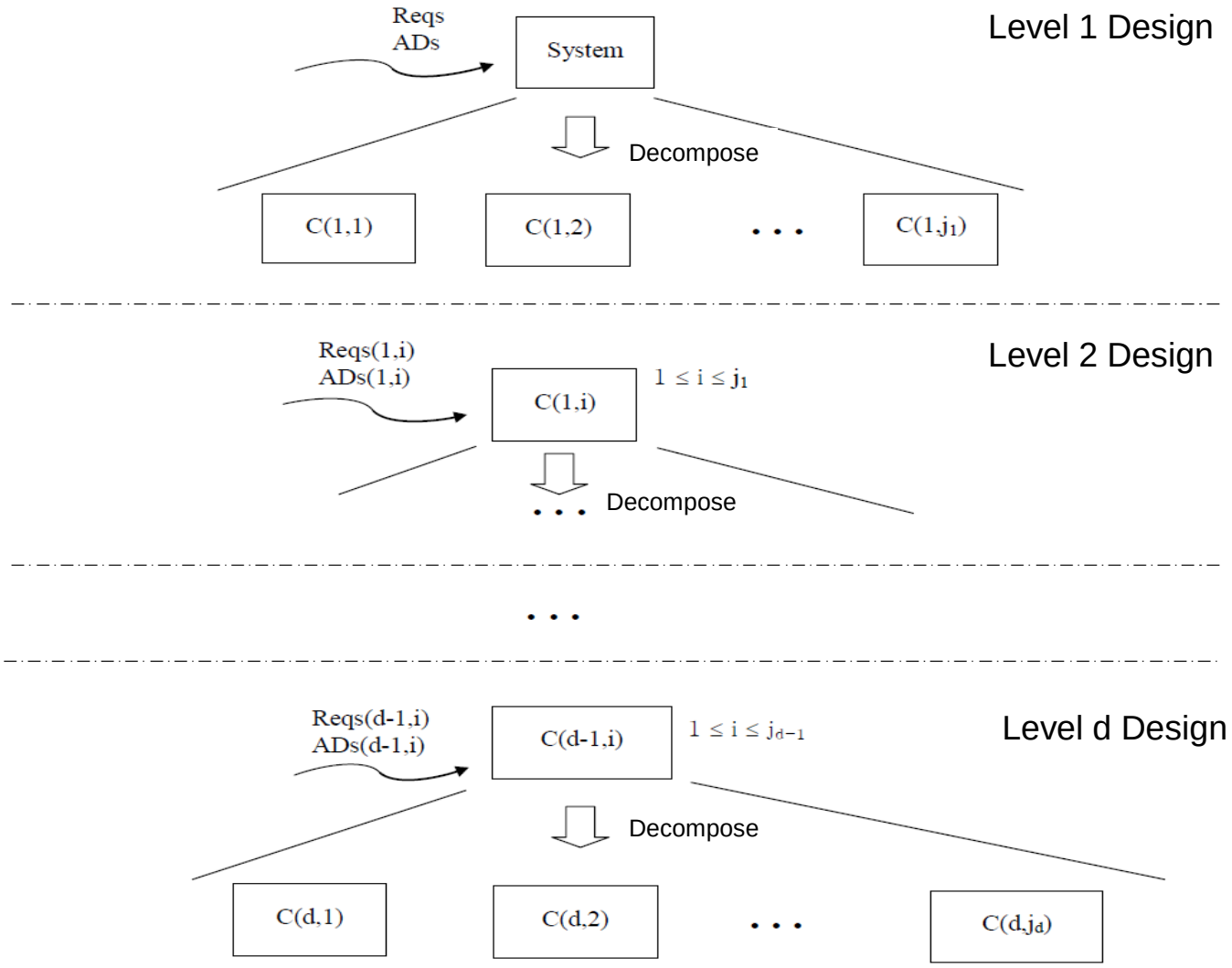
Chapter 13 (Supplementary)

1. Architecture Design Procedure
2. Logical View and Module View
3. The MVC Pattern
4. Various Architecture Patterns

1. Architecture Design Procedure

Typical Architecture Design Procedure



Reqs: requirements

ADs: architecture drivers

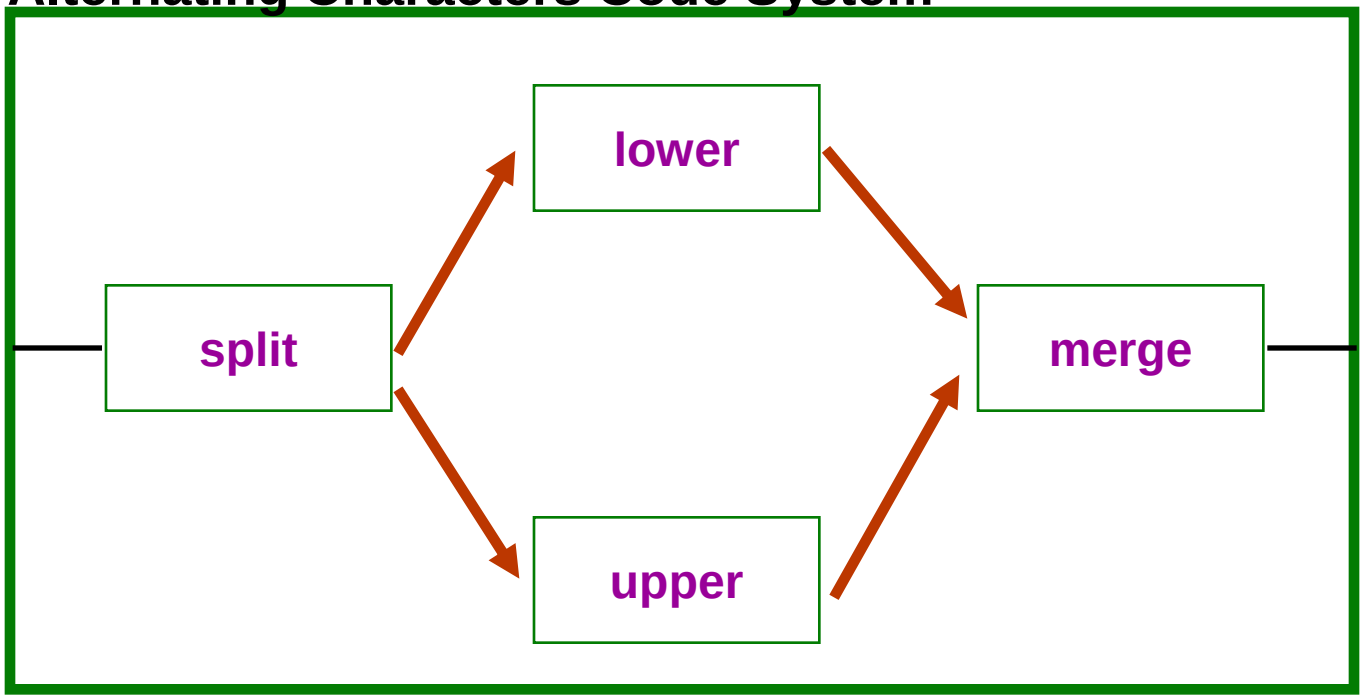
C: component

Your project may need a top-down bottom-up approach due to the use of existing components.

2. Logical View and Module View

Example - Logical View

Alternating Characters Code System



Originally called "runtime view" by D. Garlan

Produces alternating case of characters in a stream
"sofTWareArchitecture" => "SoFtWaReArChItEcTuRe"

Legend
- Filter
- Pipe
- Binding

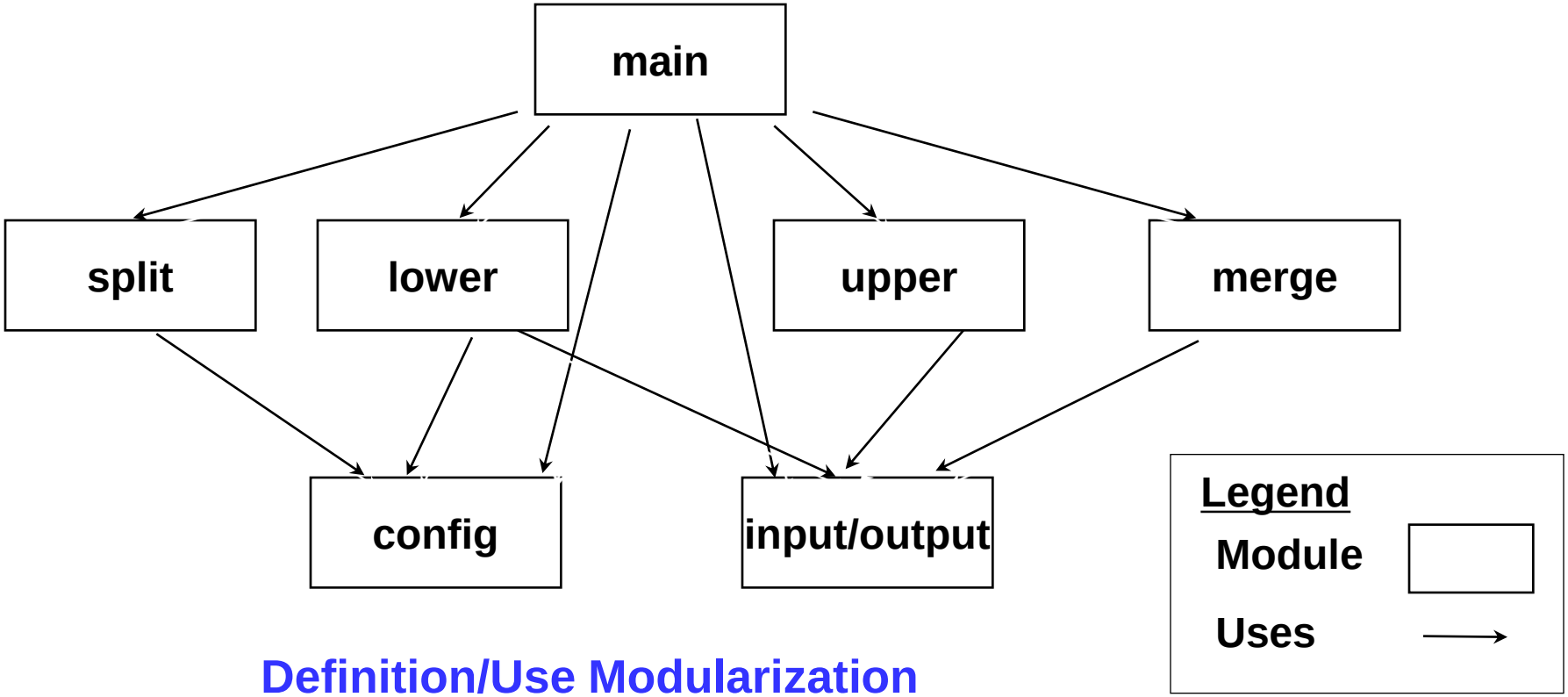
*** Source of the slide:**
David Garlan

The Logical viewpoint

- Components: Filters
- Connectors: Pipes

Example - Module View

Produces alternating case of characters in a stream



Definition/Use Modularization

* Source of the slide: David Garlan

The Module viewpoint

- Components: Modules. A module is a code unit that implements a set of responsibilities.

- Connectors: Relations among modules include
  - A is part of B. This defines a part-whole relation among modules
  - A depends on B. This defines a dependency relation among modules.
  - A is a B. This defines specialization and generalization relations among modules.

3. The MVC Architecture Pattern

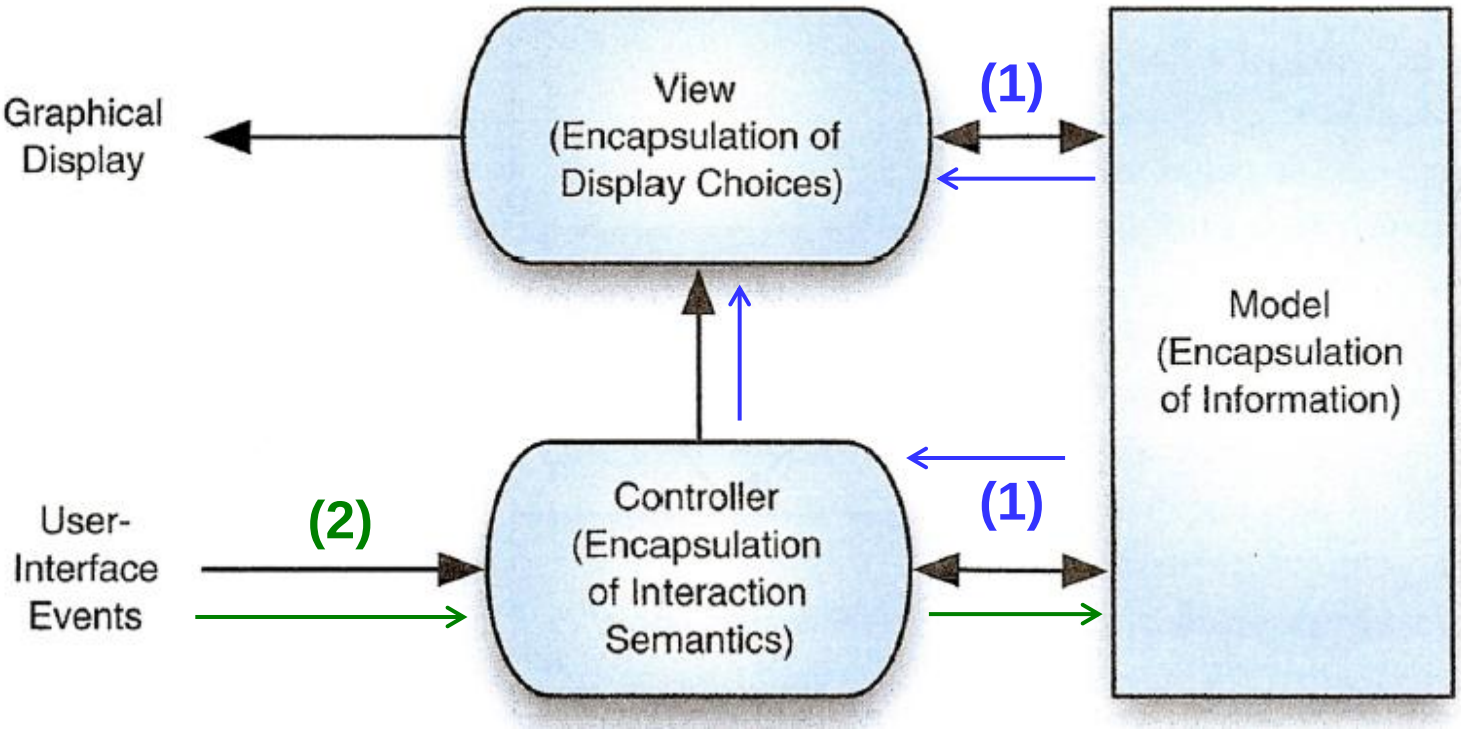
Model-View-Controller (MVC) Pattern(1/4)

- A dominant GUI design pattern since invented in 1980's
- Can be regarded as design pattern ***or*** architectural pattern
- Objectives:
  - promotes separation of information manipulated by a program and depictions of user interactions with that information

    => independent development paths for different depictions
- **Model component**: encapsulates the information used by the application
- **View component**: encapsulates the information chosen and necessary for graphical depiction of that information
- **Controller component**: encapsulates the logic necessary
  - to maintain consistency between the model and the view, and
  - to handle inputs from the user as they relate to the depiction

Model-View-Controller (MVC) Pattern (2/4)

Figure 4-4.
Notional model-view-controller pattern.



Model-View-Controller (MVC) Pattern (3/4)

- Notional **interaction** between these components:

  (1) When the application changes value in the model object, notification of that change is sent **to the view** so that any affected parts of the depiction can be updated and redrawn

    - Notification also typically goes to the controller as well, so that the controller can modify the view if its logic so requires.

  (2) When handling input from the user (such as a mouse click on part of the view), the viewing system sends the user event **to the controller**.

    - The controller then updates the model object in keeping with the desired semantics.

Model-View-Controller (MVC) Pattern (4/4)

- Can be seen at work in the WWW.
  - Web resources –> model
  - HTML rendering agent within a browser –> view
  - Part of the browser that responds to user input and which causes either interactions with a Web server or modifies the browser's display –> controller

- Inspired the development of PAC (Presentation-Abstraction-Control) pattern

4. 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

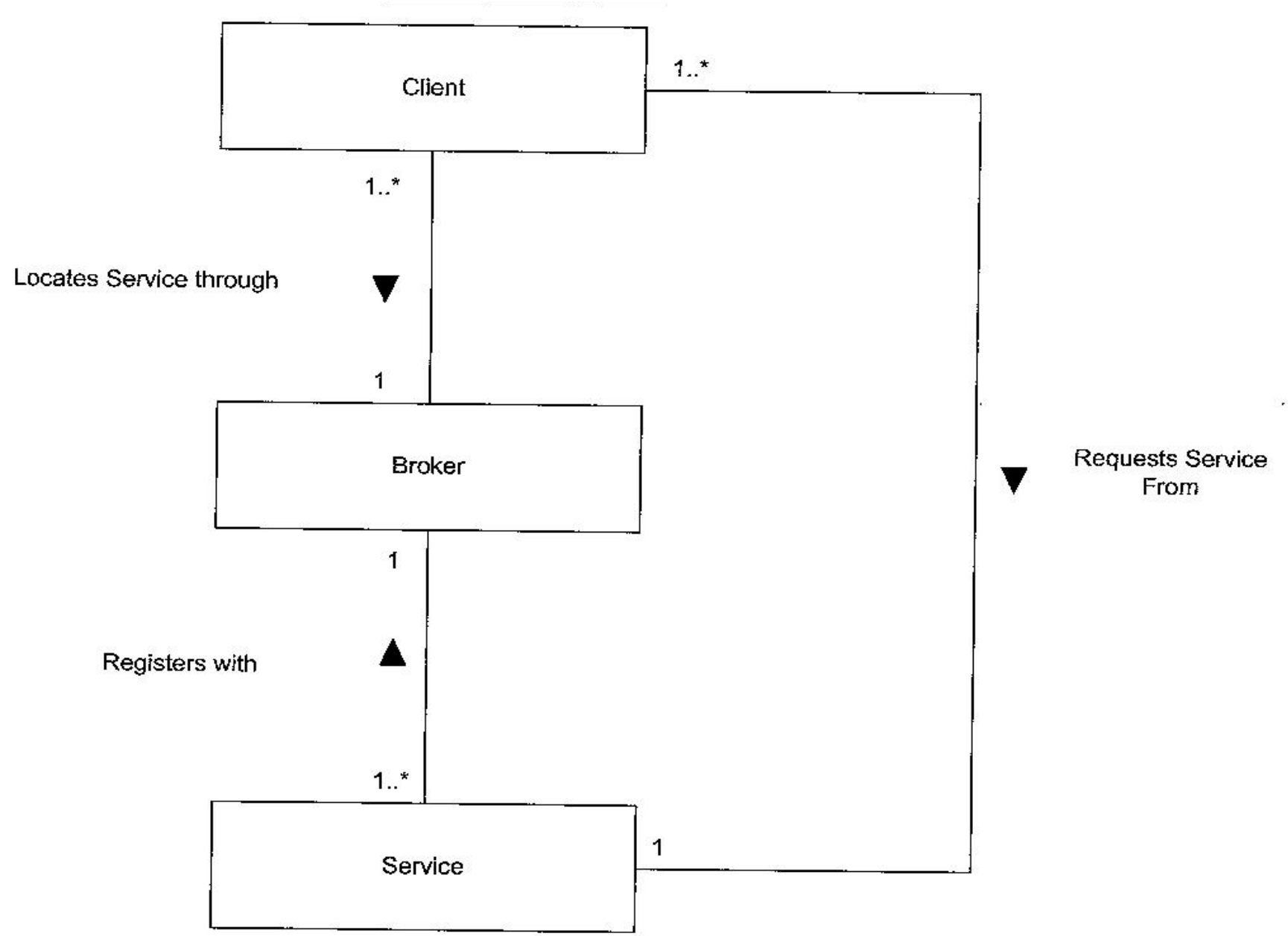


Figure A.1. Broker pattern

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

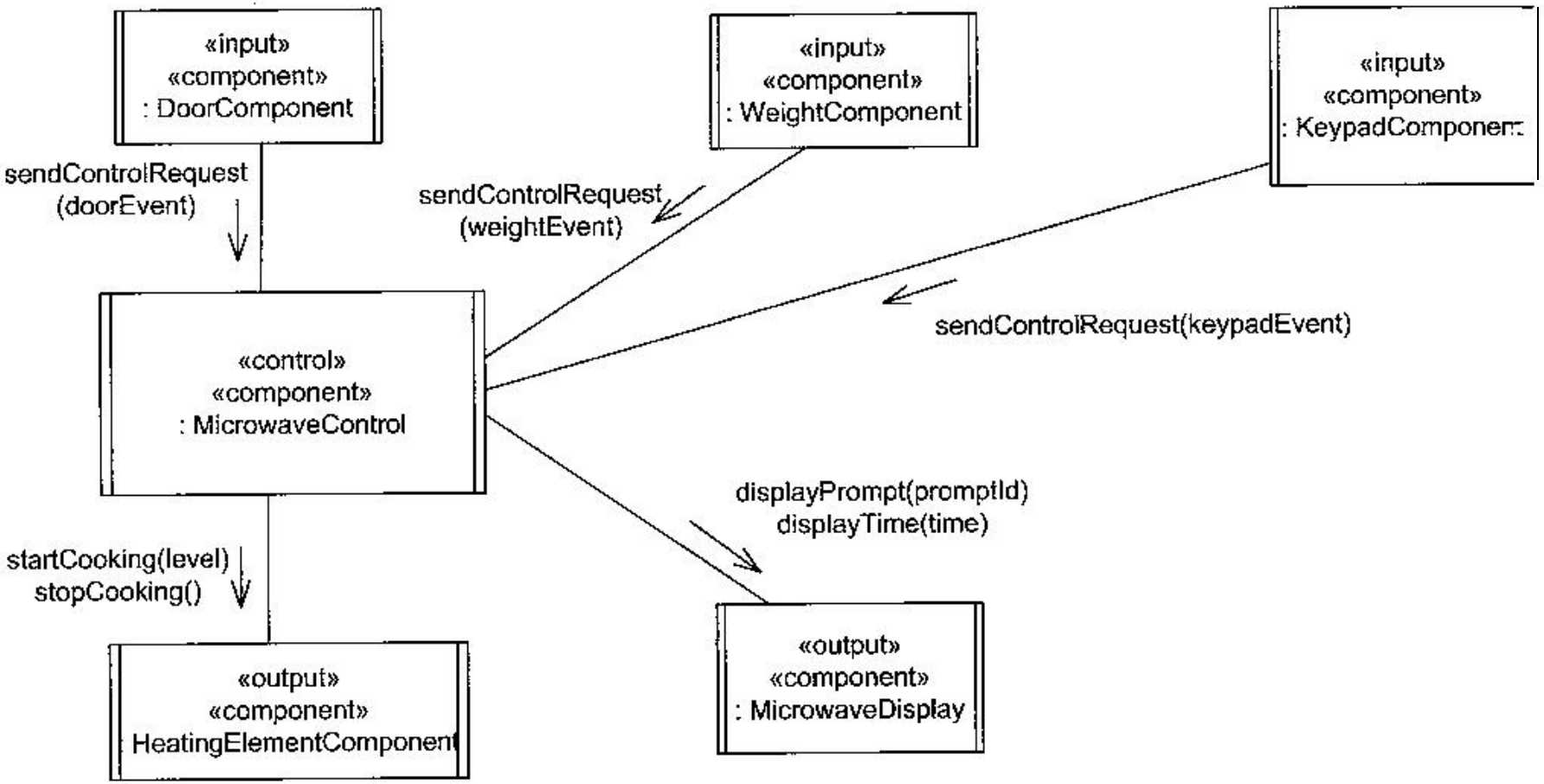


Figure A.2. Centralized Control pattern: Microwave Oven Control System example

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

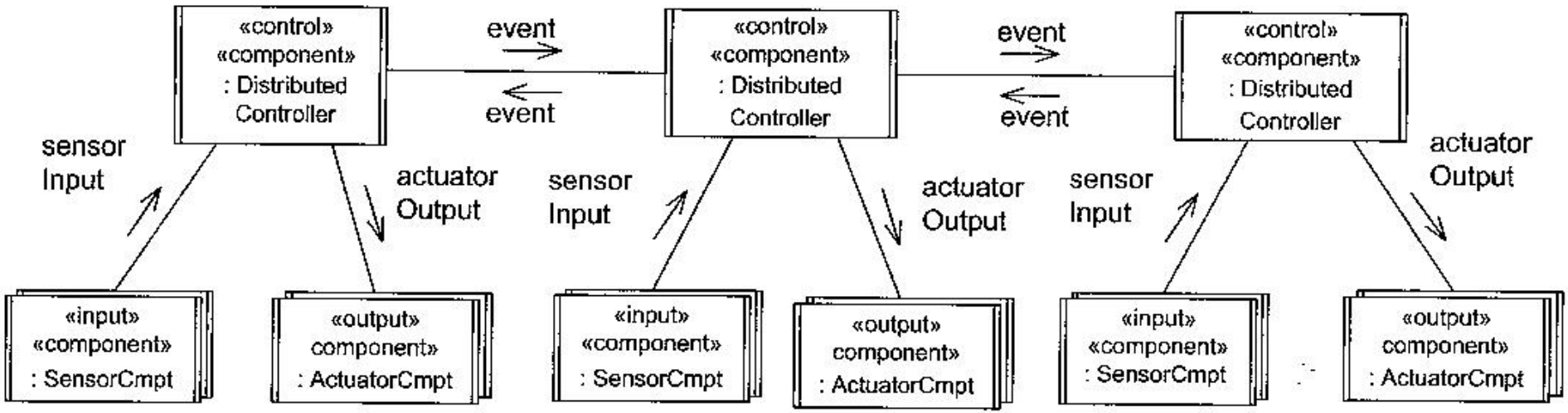


Figure A.3. Distributed Control pattern

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

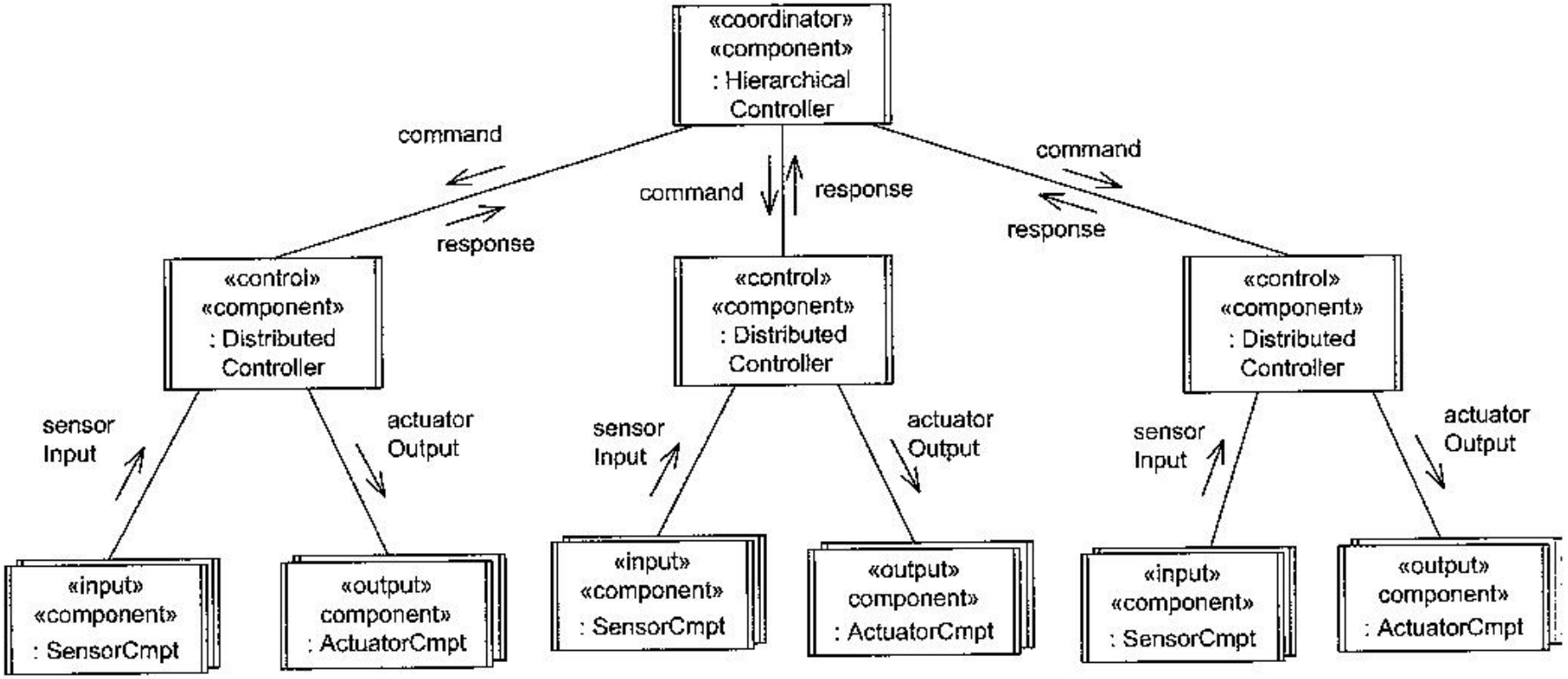


Figure A.4. Hierarchical Control pattern

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

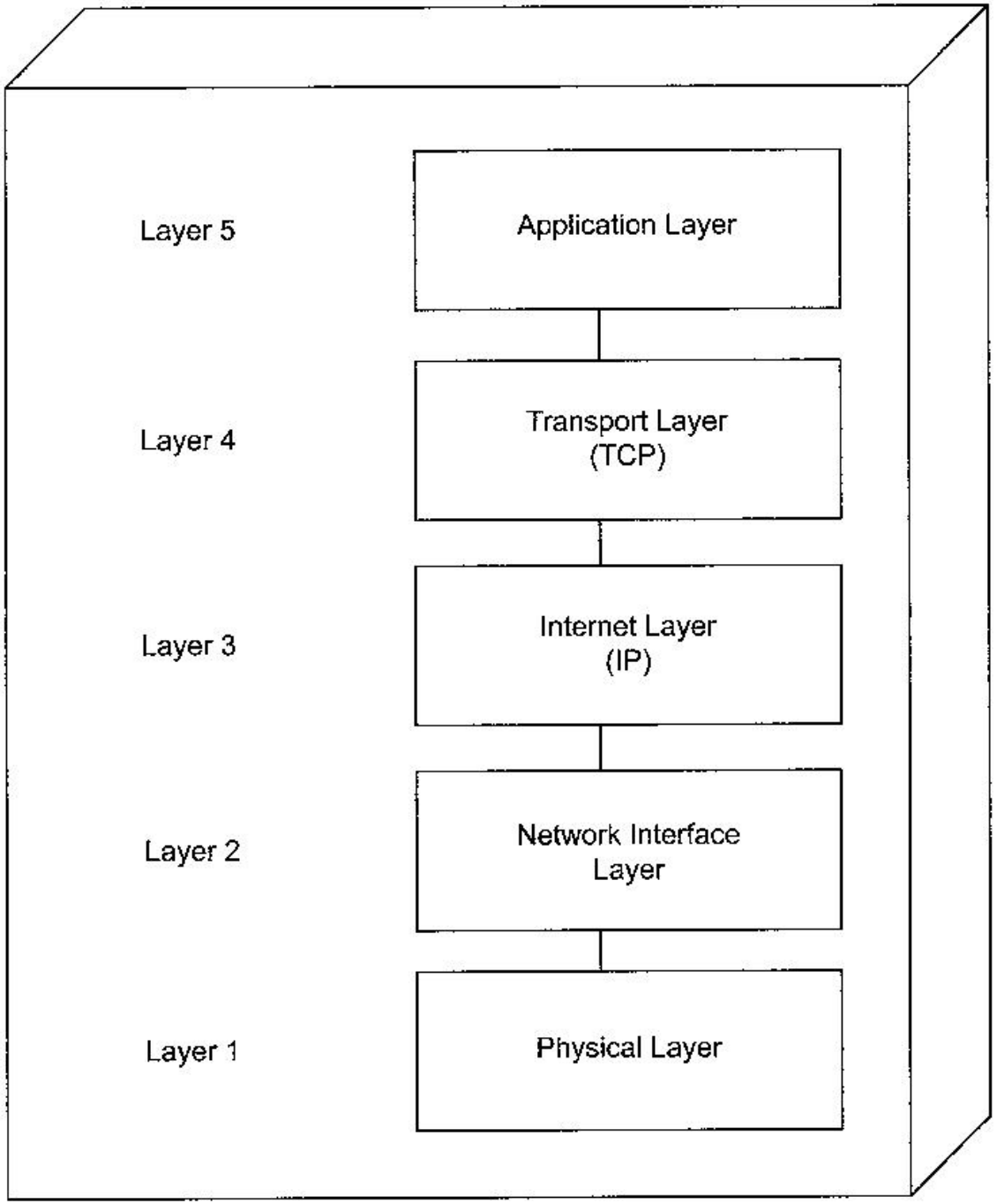


Figure A.5. Layers of Abstraction pattern: TCP/IP example

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

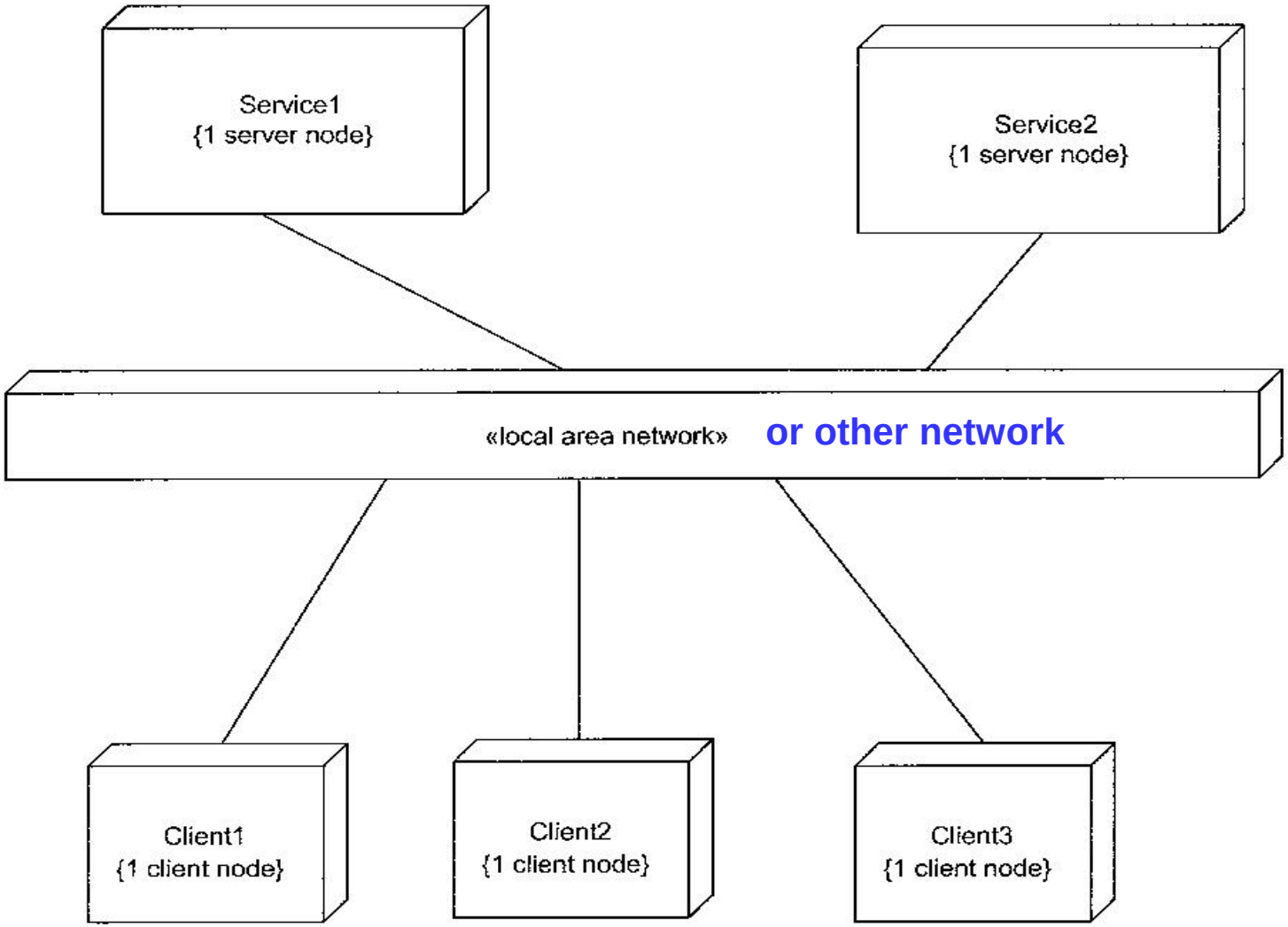


Figure A.6. Multiple Client/Multiple Service Pattern

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

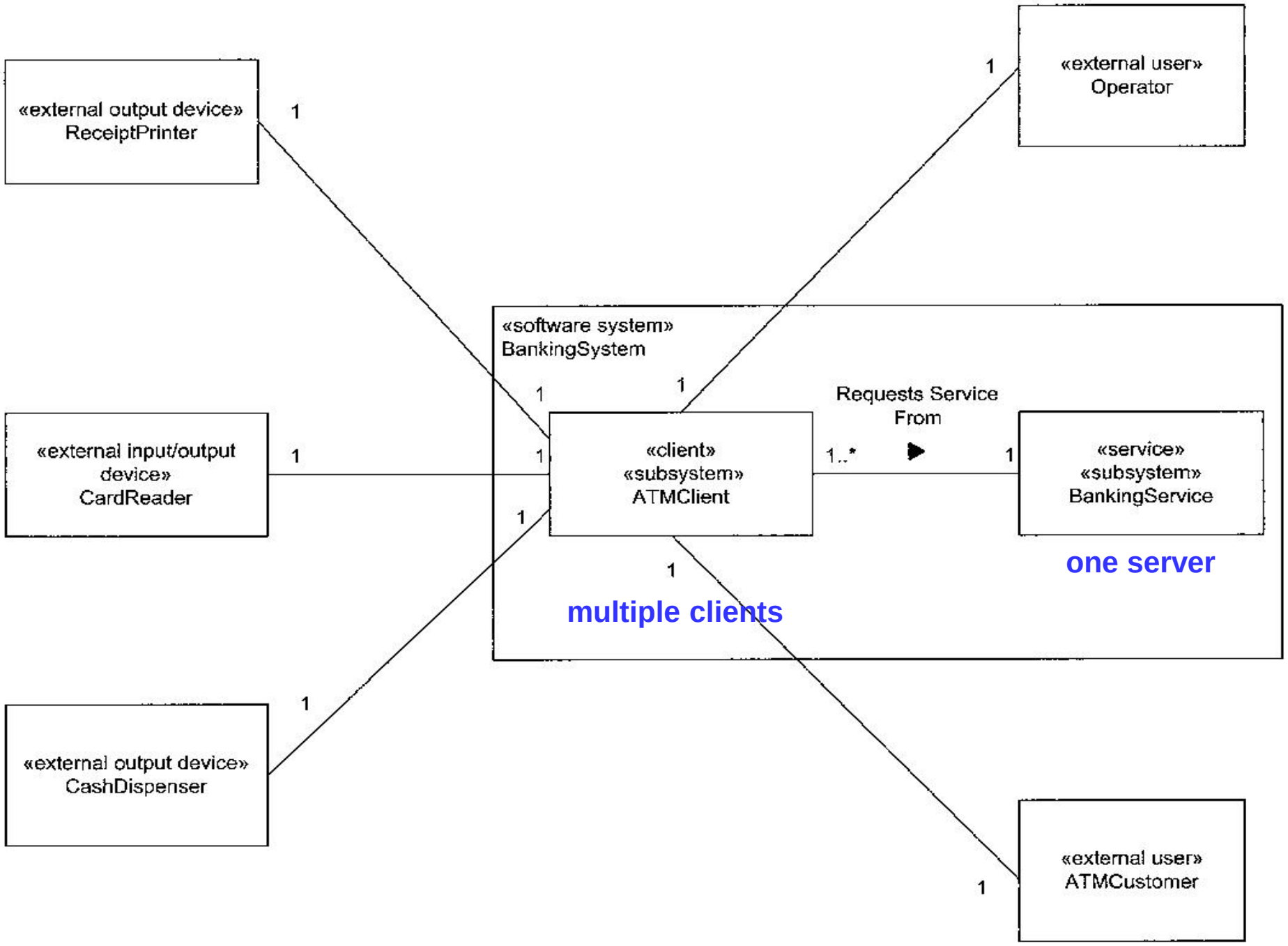


Figure A.7. Multiple Client/Single Service Pattern: Banking System example

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

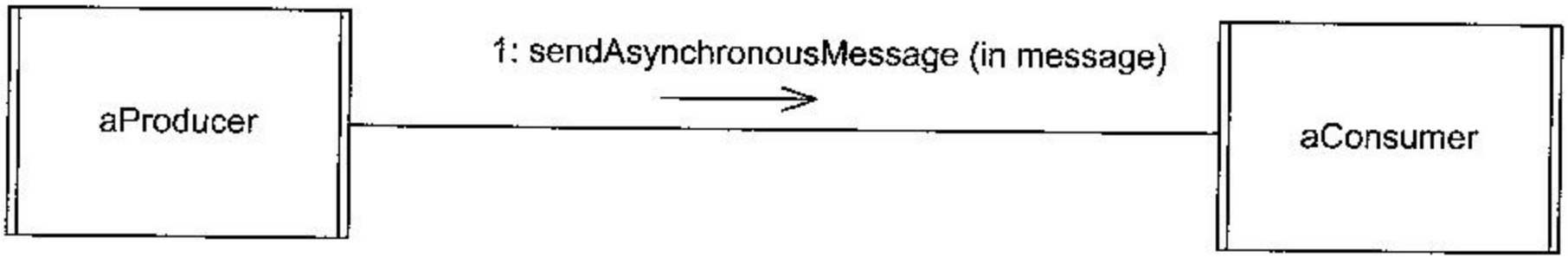


Figure A.9. Asynchronous Message Communication pattern

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

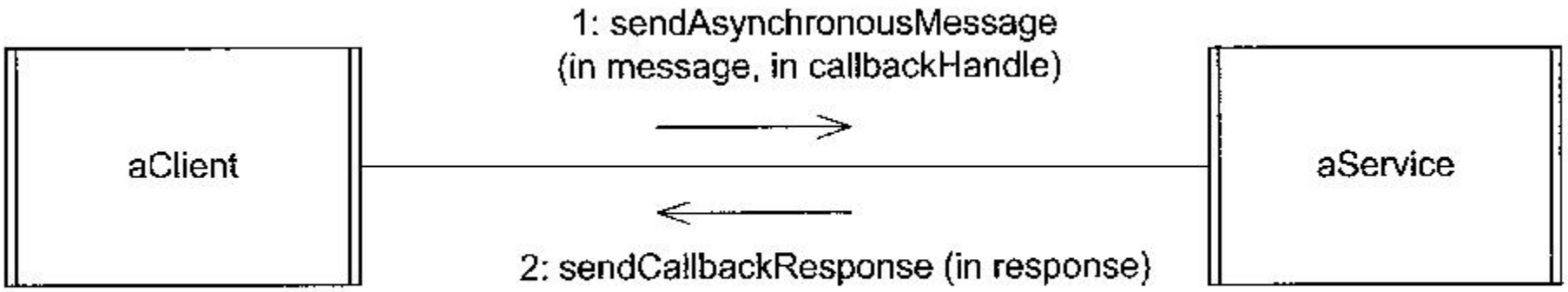


Figure A.10. Asynchronous Message Communication with Callback pattern

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

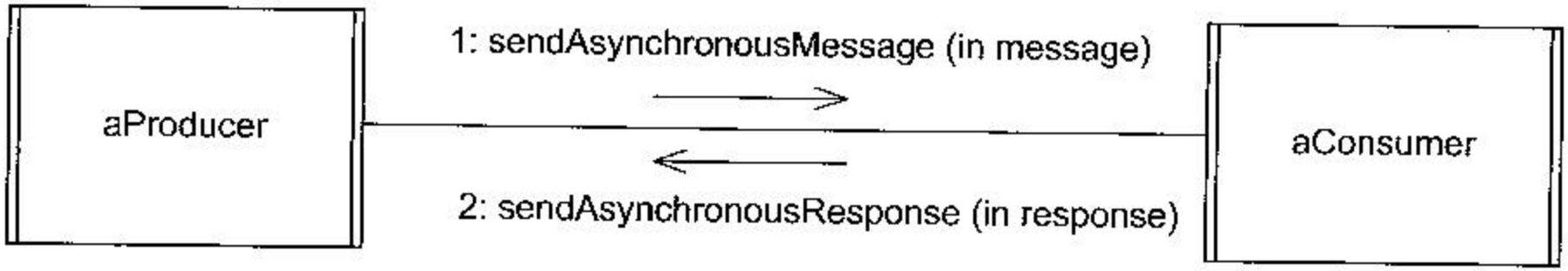


Figure A.11. Bidirectional Asynchronous Message Communication pattern

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

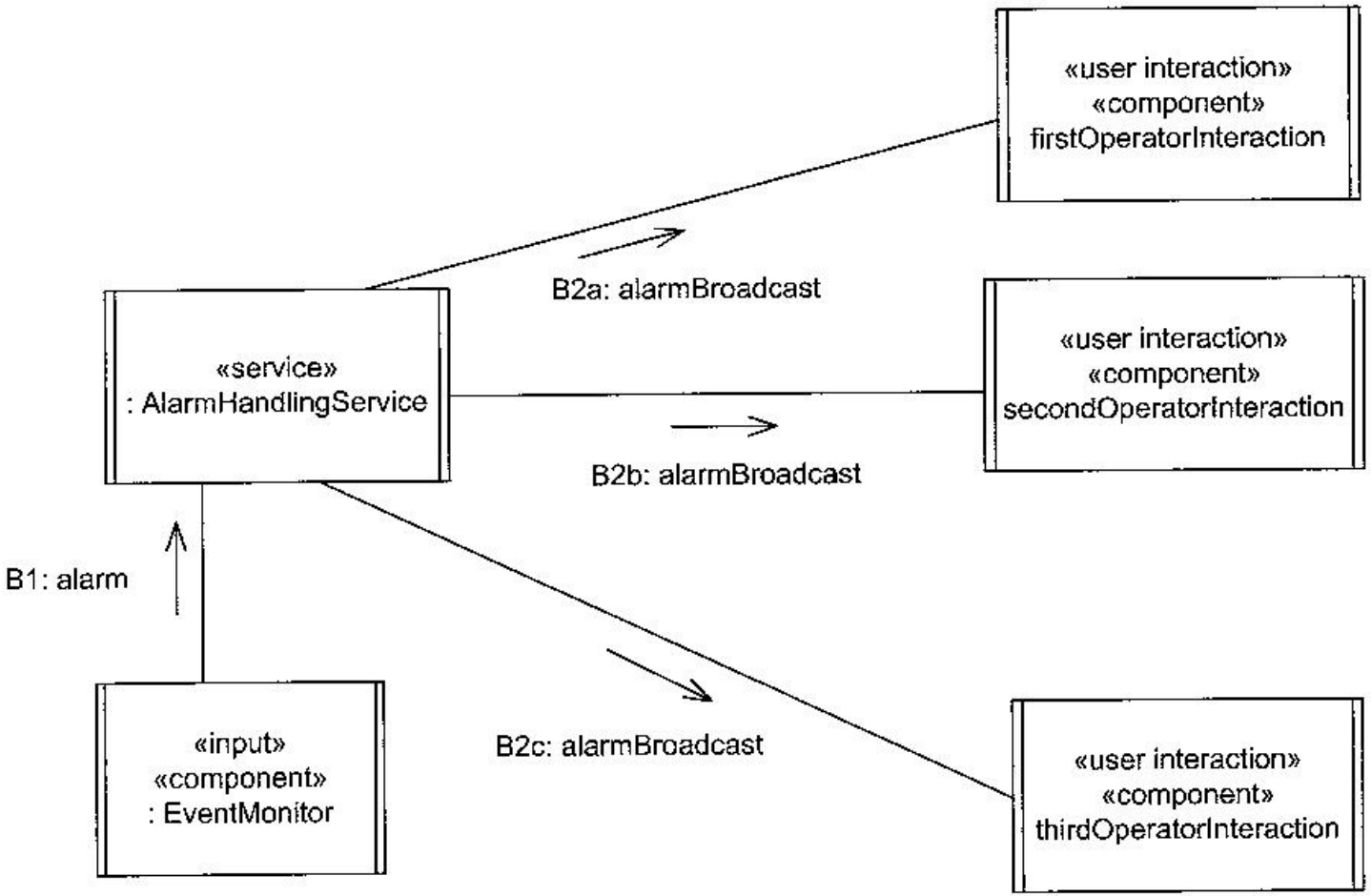


Figure A.12. Broadcast pattern: alarm broadcast example

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

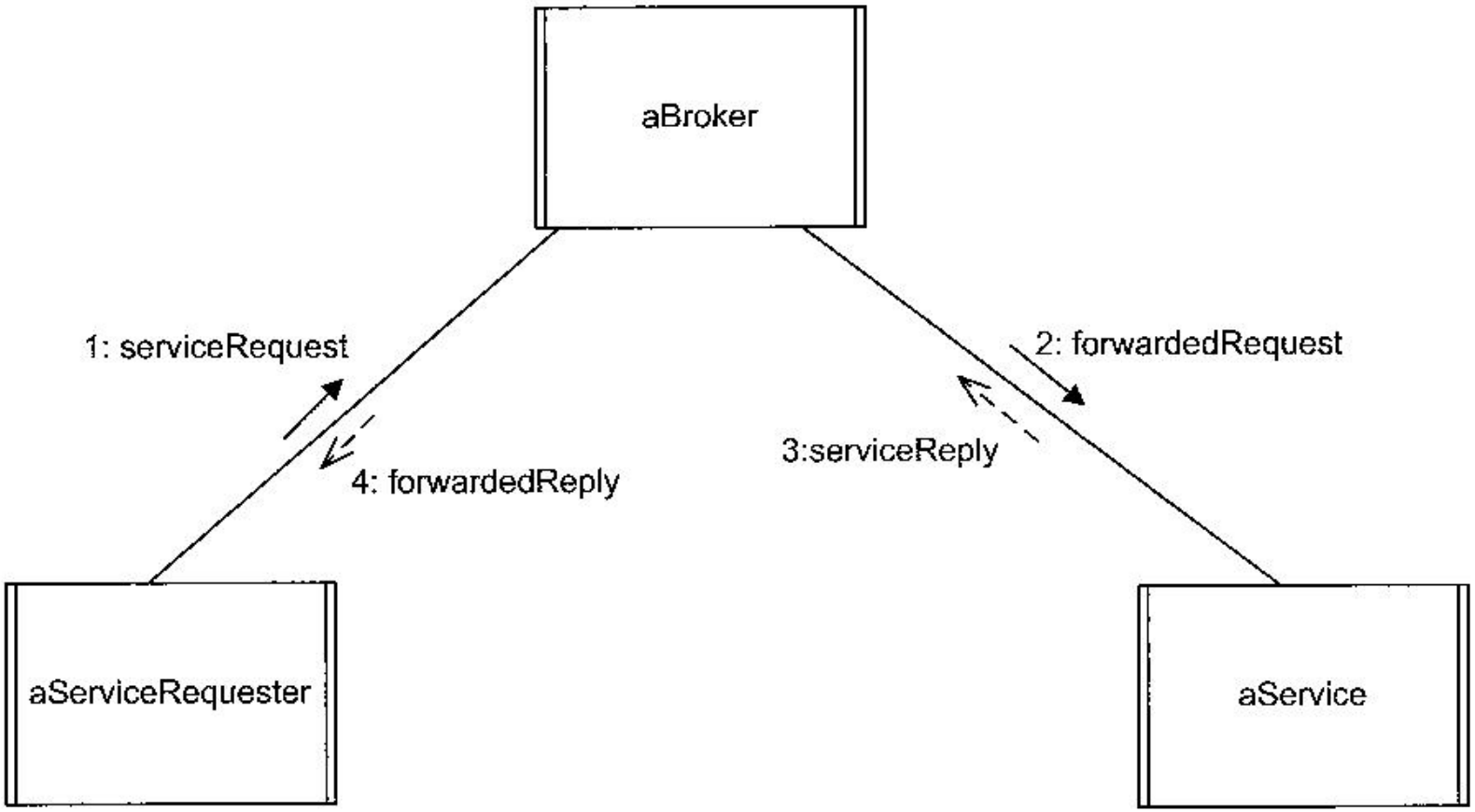


Figure A.13. Broker Forwarding pattern

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

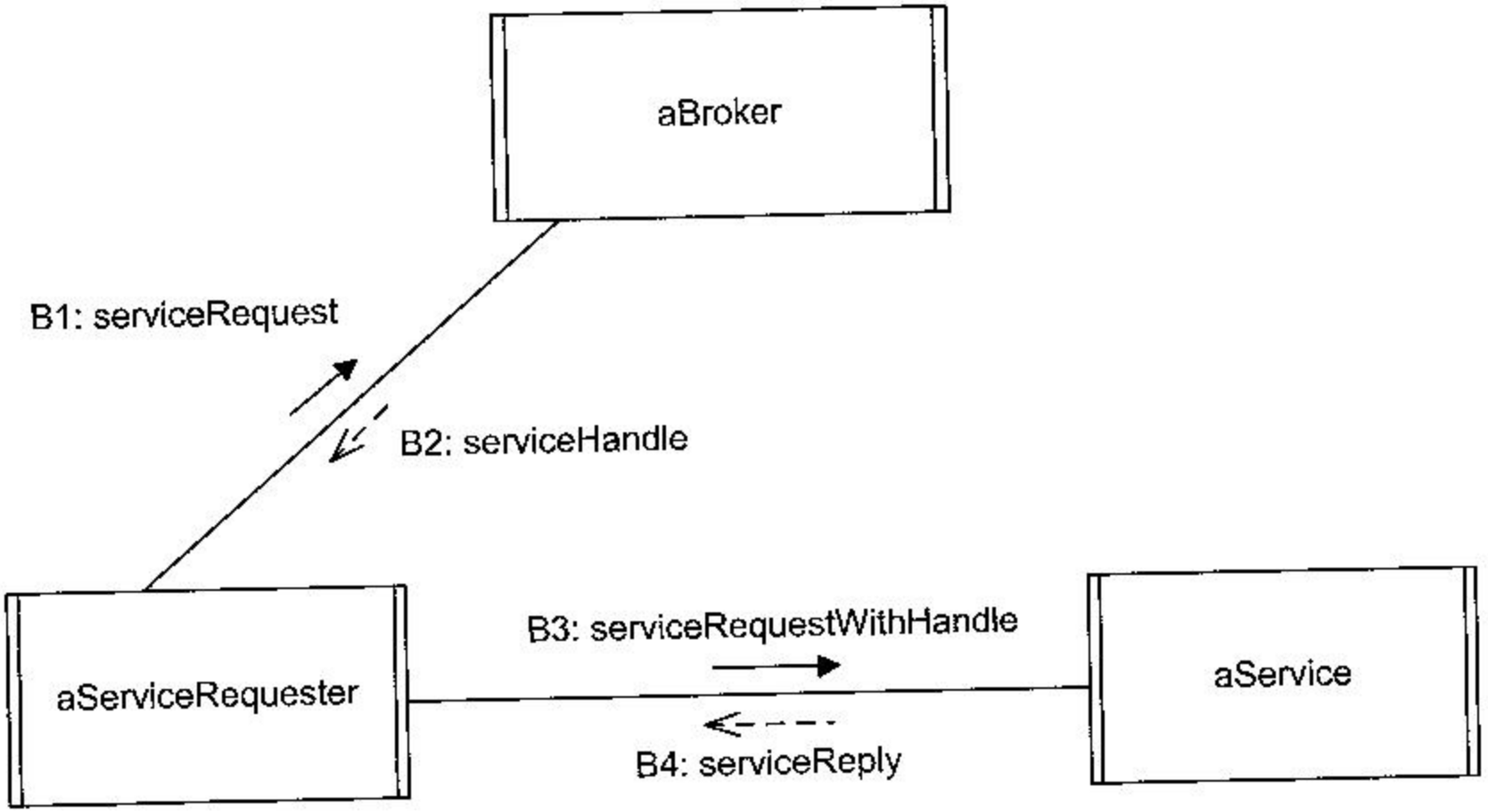


Figure A.14. Broker Handle pattern

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

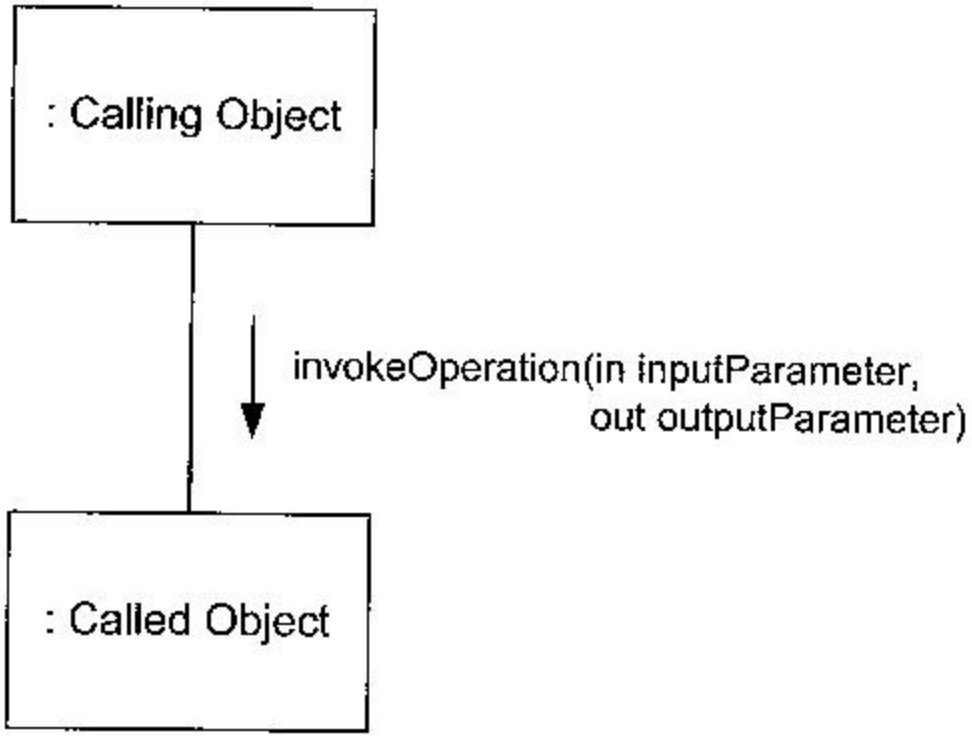


Figure A.15. Call/Return pattern

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

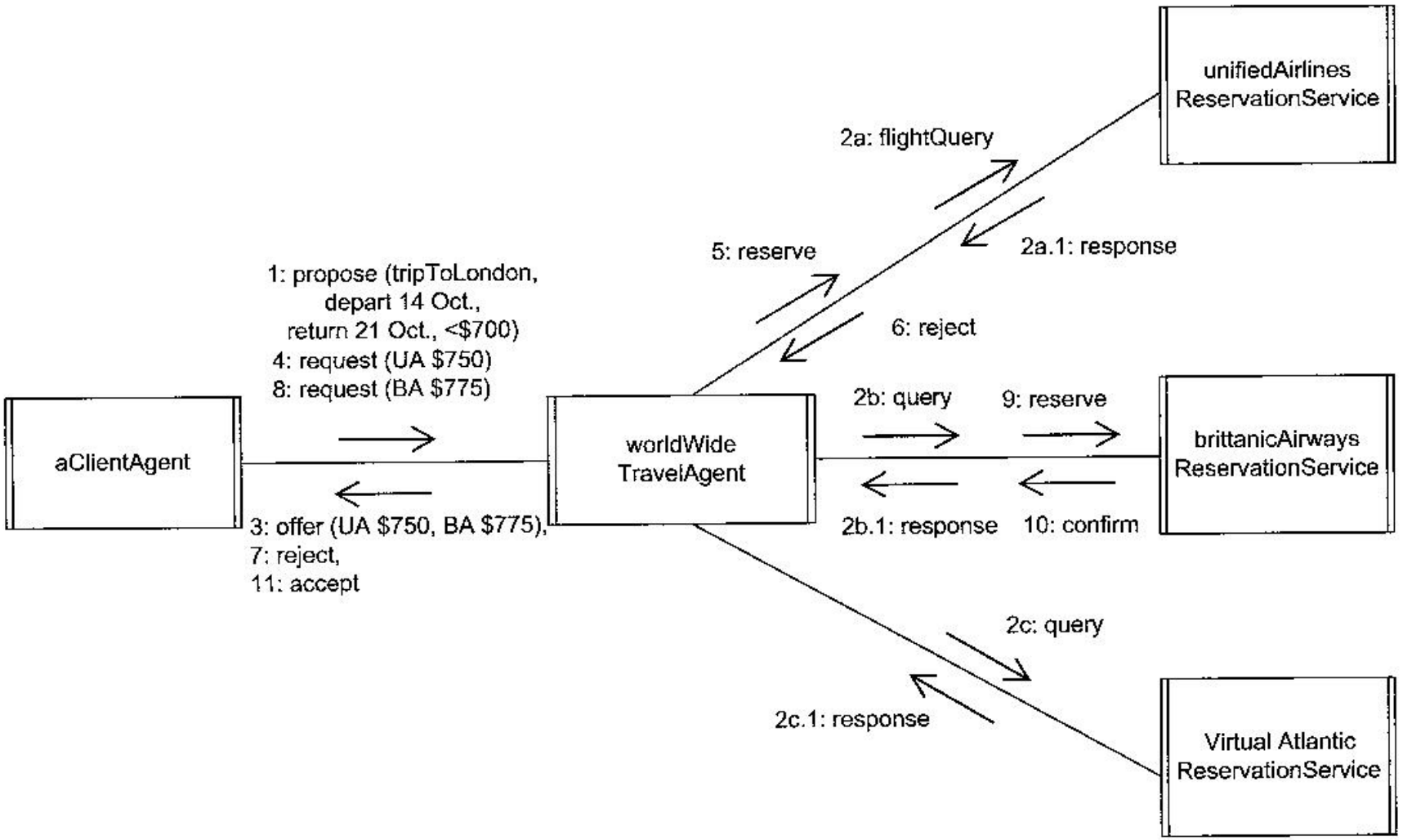


Figure A.16. Negotiation pattern: airline reservation example

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

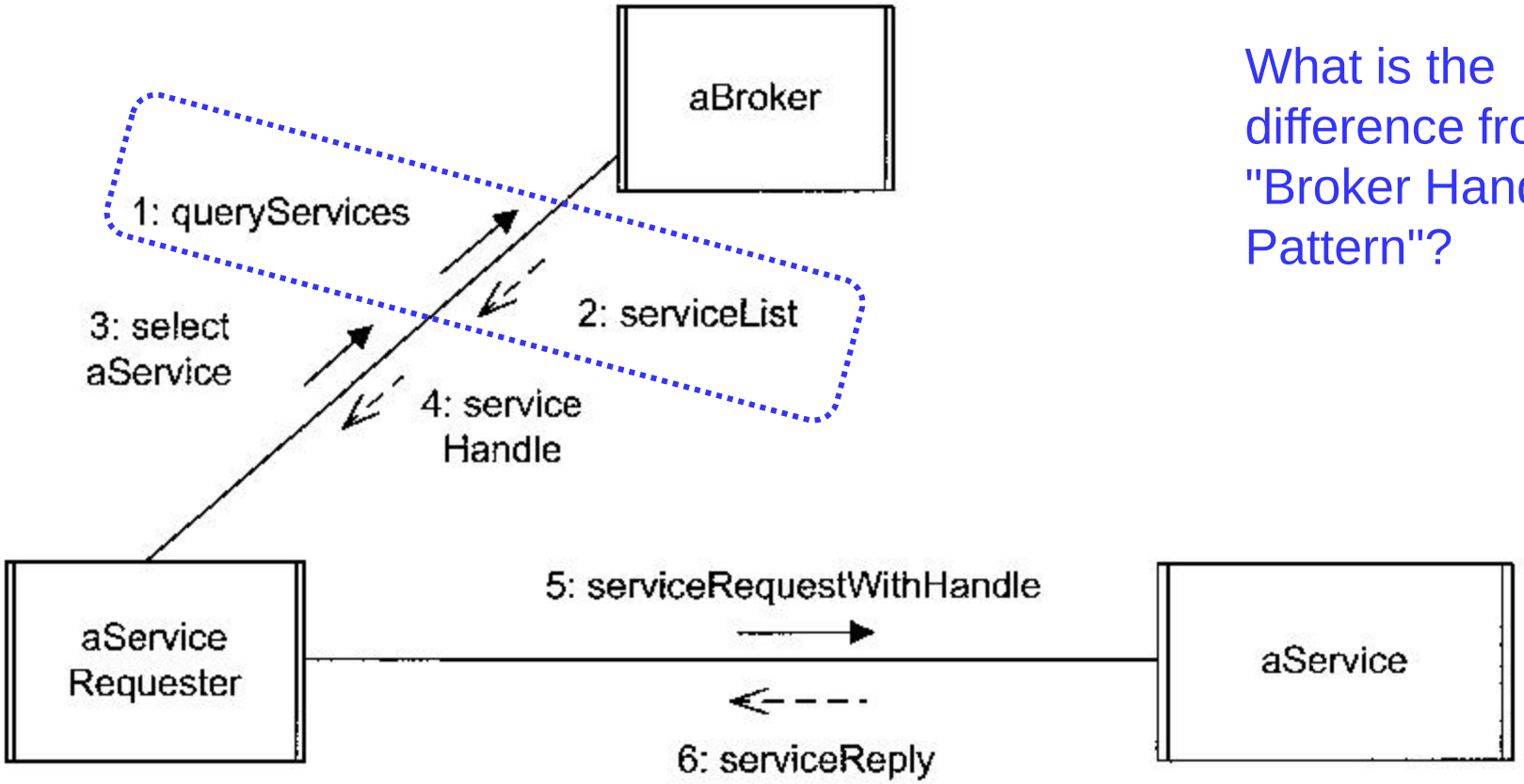


Figure A.17. Service Discovery pattern

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

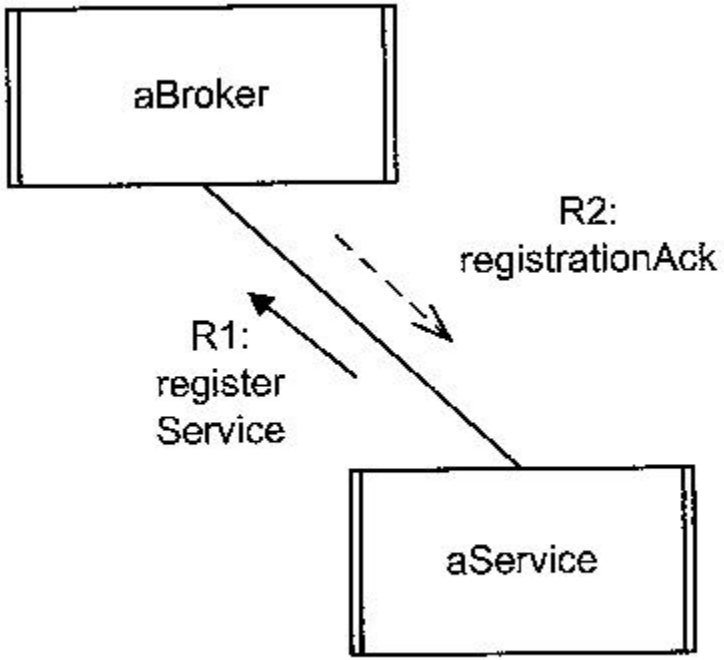


Figure A.18. Service Registration pattern

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

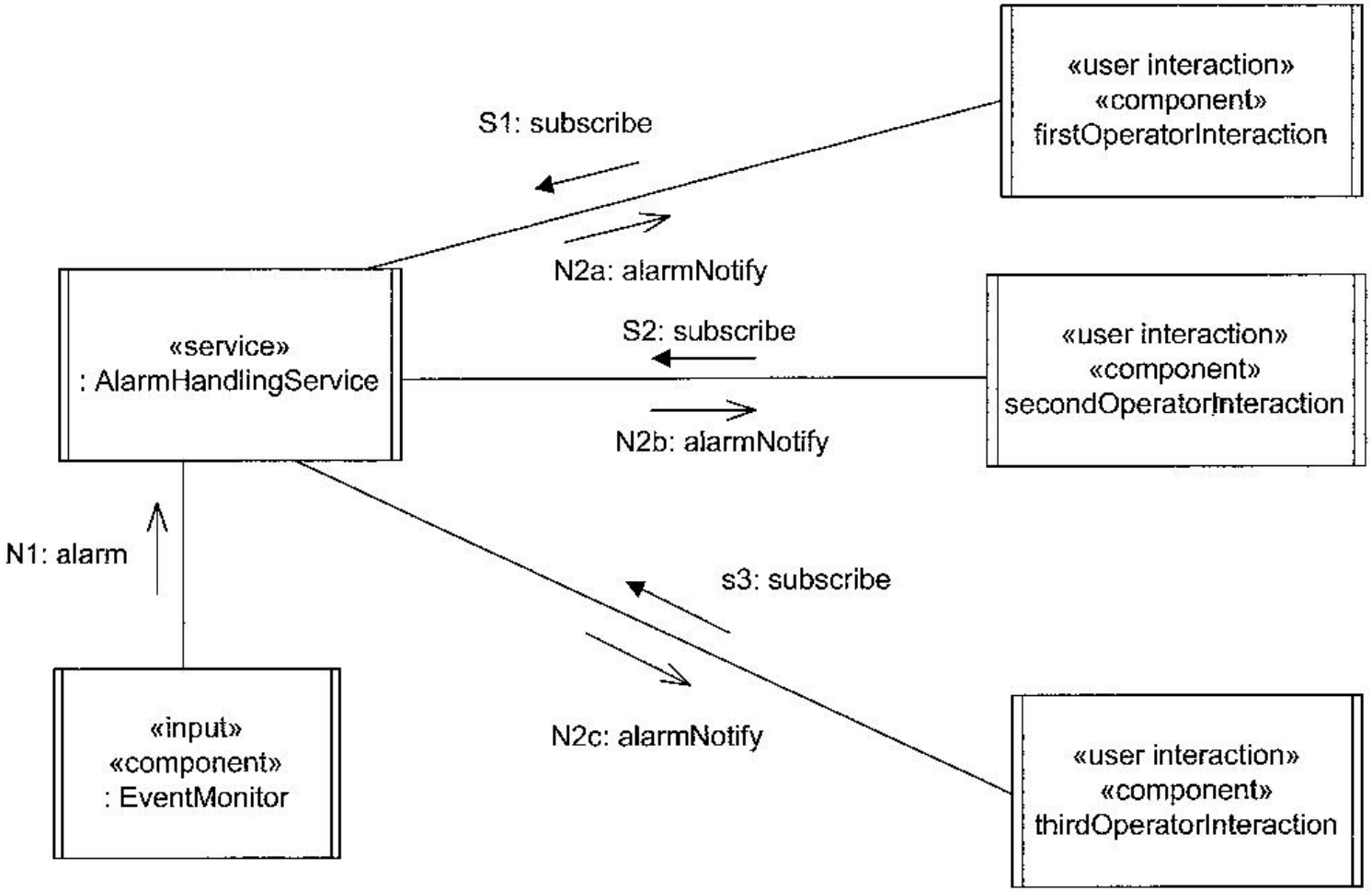


Figure A.19. Subscription/Notification pattern: alarm notification example

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

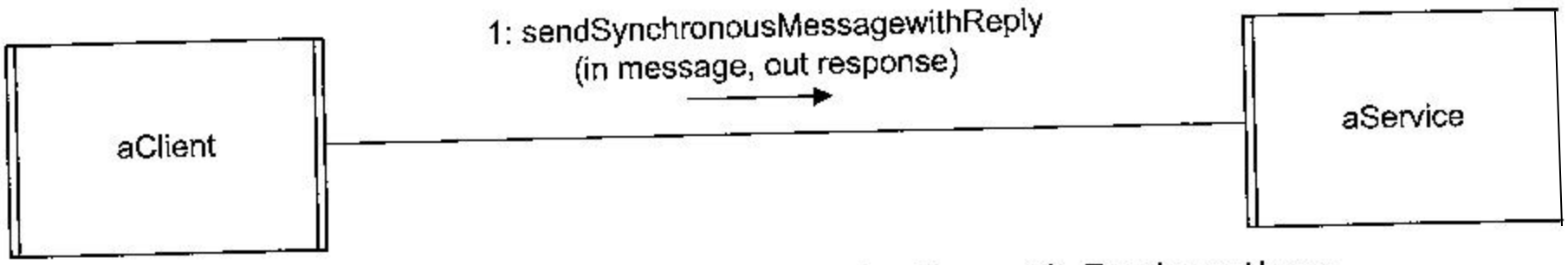


Figure A.20. Synchronous Message Communication with Reply pattern

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

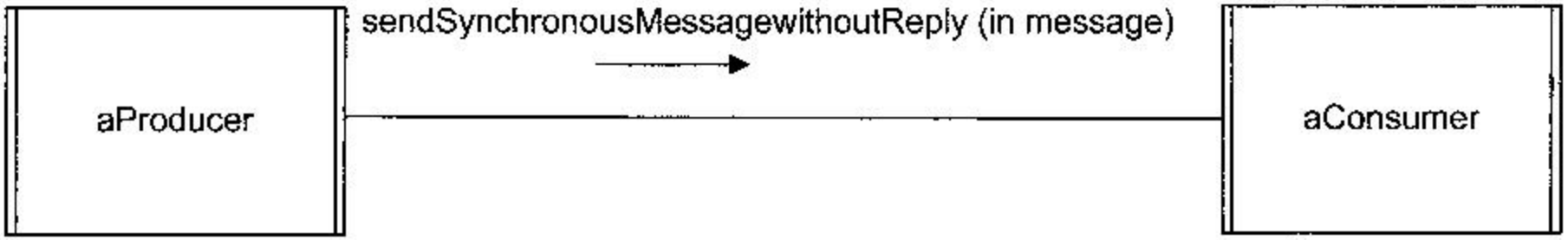


Figure A.21. Synchronous Message Communication without Reply pattern

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

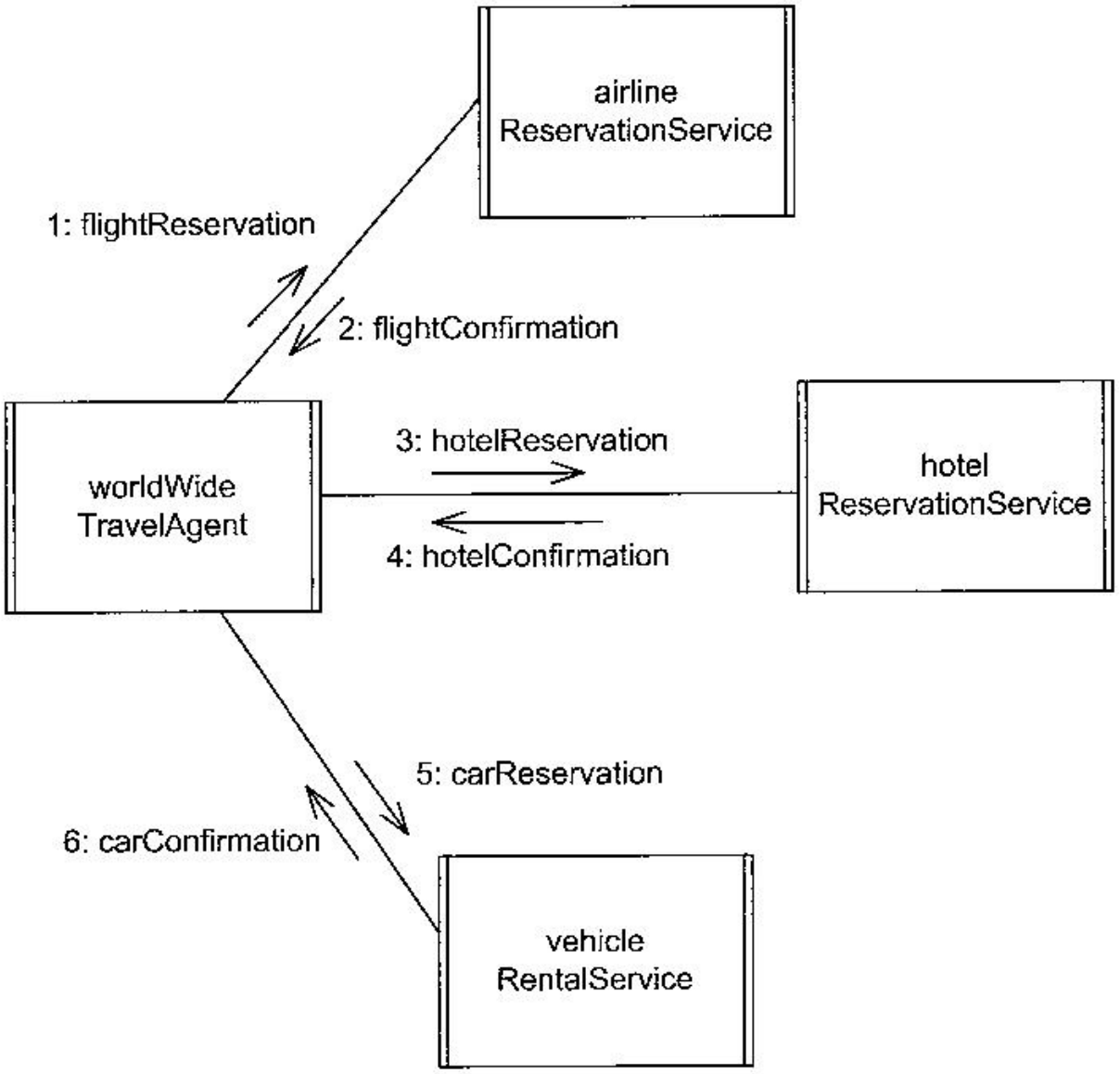


Figure A.22. Compound Transaction pattern: airline/hotel/car reservation example

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

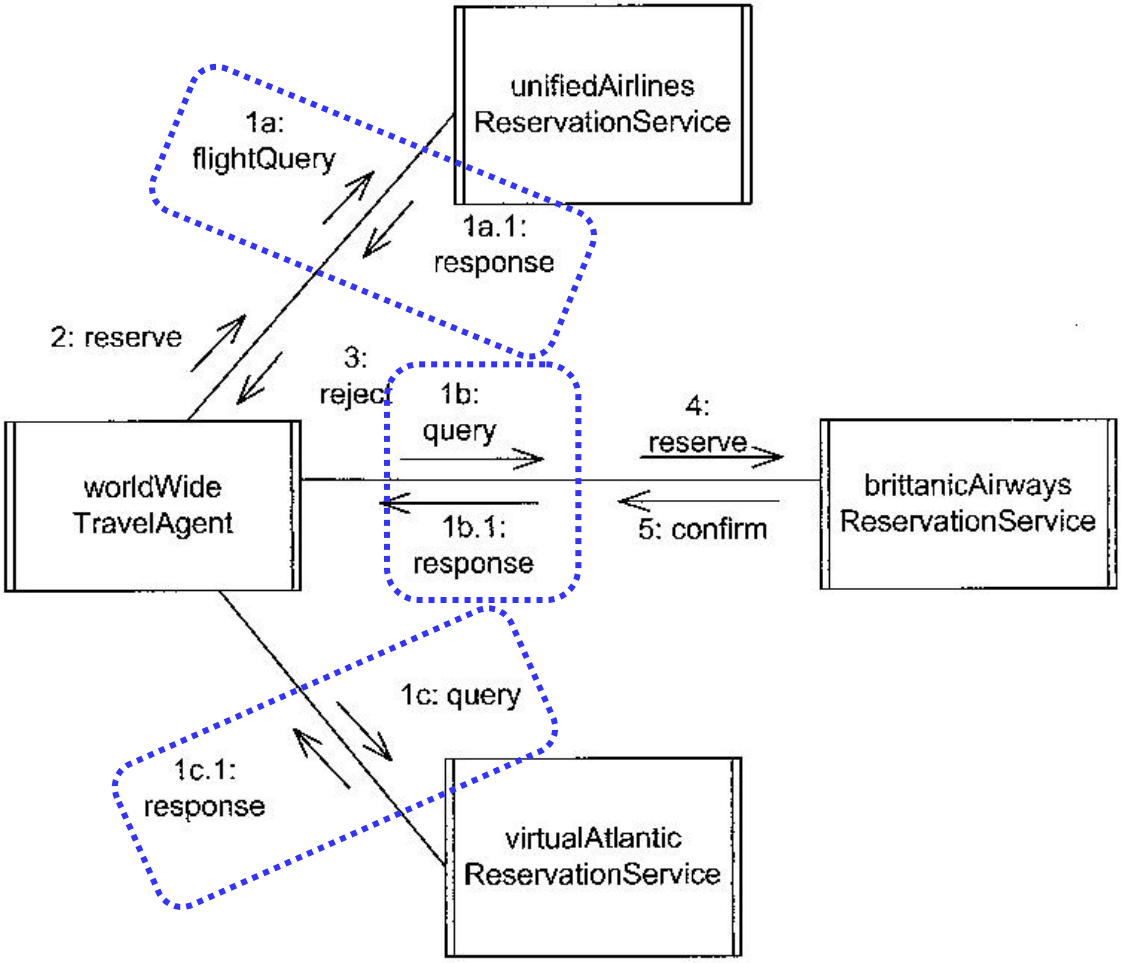
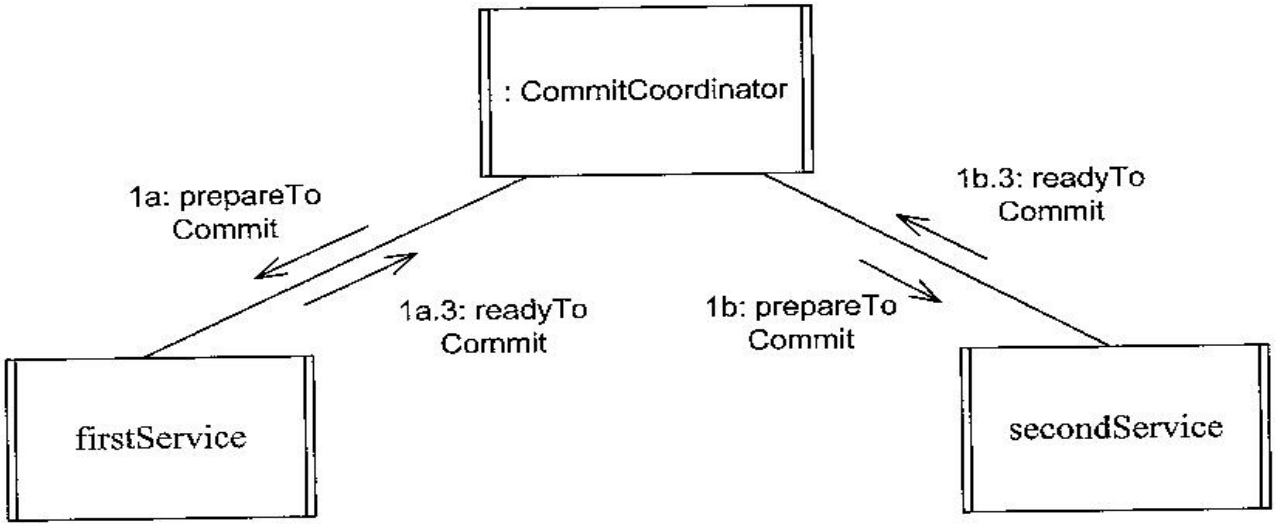


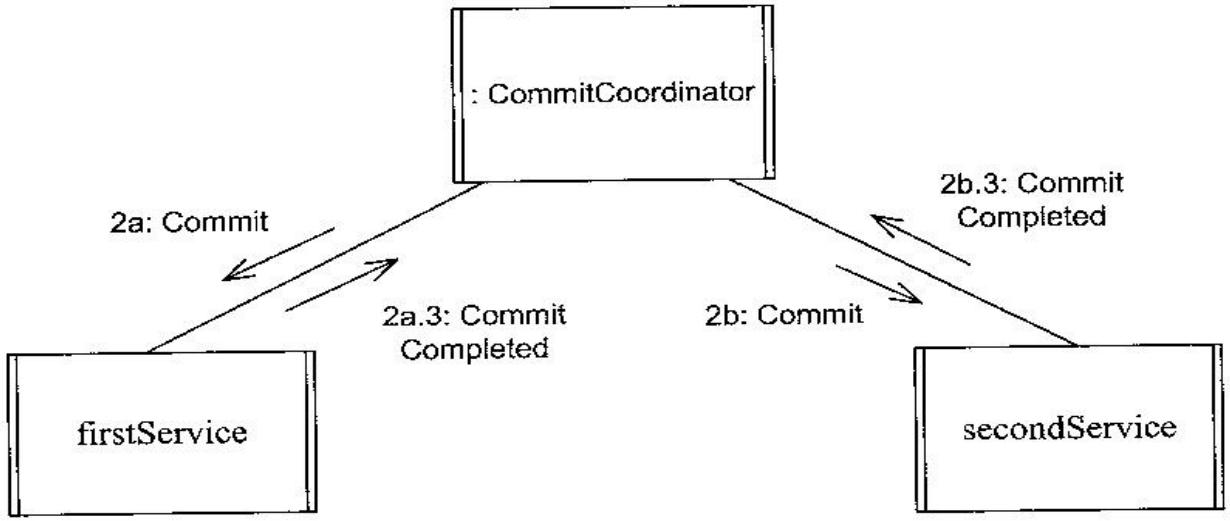
Figure A.23. Long-Living Transaction pattern: airline reservation example

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?