Data-Flow Architecture : *Pipe and Filter*

The *Pipes and Filters* architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

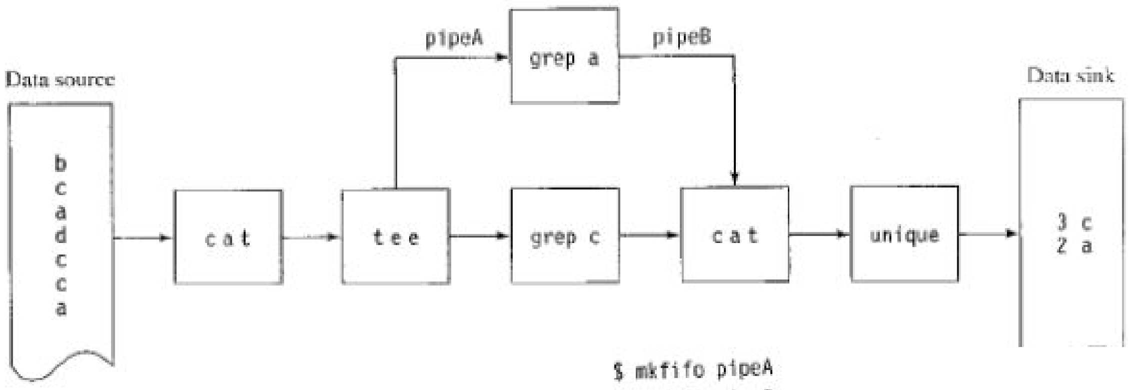
Context Processing data streams.

Problem Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons: the system has to be built by several developers, the global system task decomposes naturally into several processing stages, and the requirements are likely to change.

You therefore plan for future flexibility by exchanging or reordering the processing steps. By incorporating such flexibility, it is possible to build a family of systems using existing processing components. The design of the system—especially the interconnection of processing steps—has to consider the following *forces*:

- Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
- Small processing steps are easier to reuse in different contexts than large components.
- Non-adjacent processing steps do not share information.
- Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings, for example.
- It should be possible to present or store final results in various ways.
- Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.
- You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.

Solution The Pipes and Filters architectural pattern divides the task of a system into several sequential processing steps. These steps are connected by the data flow through the system—the output data of a step is the input to the subsequent step. Each processing step is implemented by a *filter* component. A filter consumes and delivers data incrementally—in contrast to consuming all its input before producing any output—to achieve low latency and enable real parallel processing. The input to the system is provided by a *data source* such as a text file. The output flows into a *data sink* such as a file, terminal, animation program and so on. The data source, the filters and the data sink are connected sequentially by *pipes*. Each pipe implements the data flow between adjacent processing steps. The sequence of filters combined by pipes is called a *processing pipeline*.



```
$ mkfifo pipeA
$ mkfifo pipeB
$ grep a < pipeA >pipeB &
$ cat infile | tee pipeA | grep c |cat - pipeB | uniq -c
```

Figure 5.8
Simple Unix pipe and filter architecture

Benefits:

- *Concurrency:* It provides high overall throughput for excessive data processing.
- *Reusability:* Encapsulation of filters makes it easy to plug and play, and to substitute.
- *Modifiability:* It features low coupling between filters, less impact from adding new filters, and modifying the implementation of any existing filters as long as the I/O interfaces are unchanged.
- *Simplicity:* It offers clear division between any two filters connected by a pipe.
- *Flexibility:* It supports both sequential and parallel execution.

Limitations:

- It is not suitable for dynamic interactions.
- A low common denominator is required for data transmission in the ASCII formats since filters may need to handle data streams in different formats, such as record type or XML type rather than character type.
- Overhead of data transformation among filters such as parsing is repeated in two consecutive filters.
- It can be difficult to configure a pipe and filter system dynamically.

Data-Flow Architecture : ***Process Control Architecture***

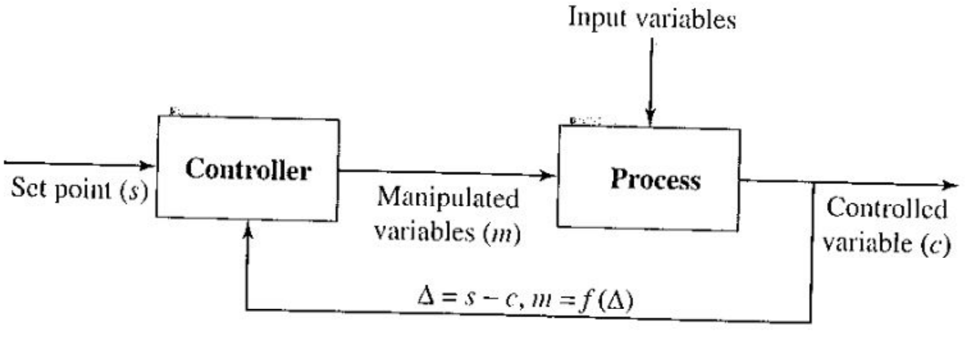
5.4 Process Control Architecture

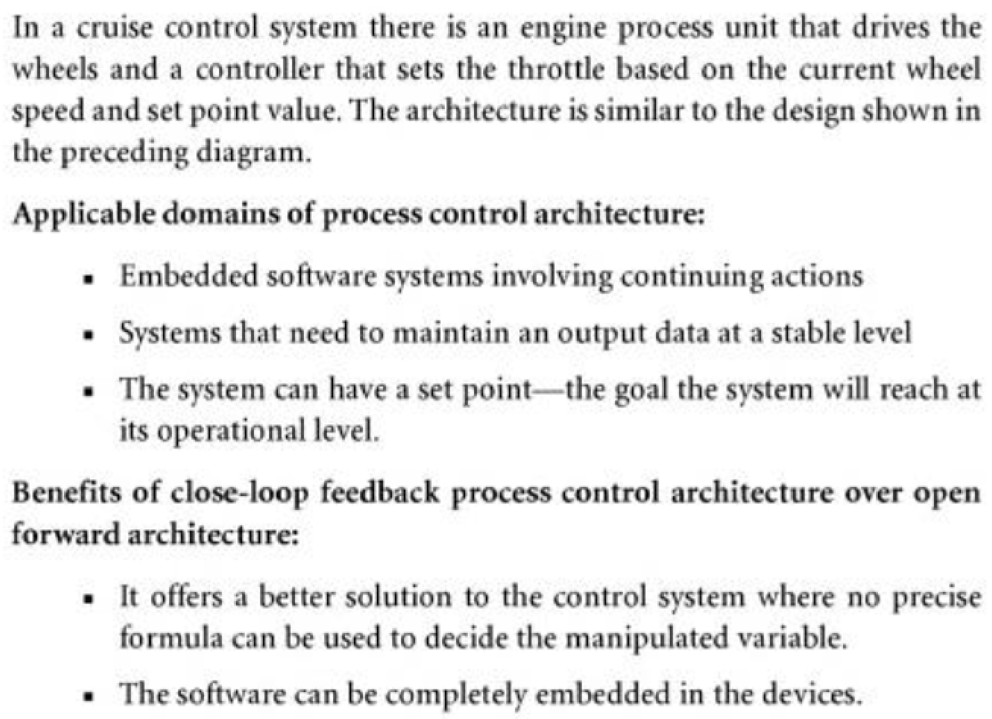
Process control software architecture is suitable for the embedded systen software design where the system is manipulated by a process control vari able data. Process control architecture decomposes the whole system intc subsystems (modules) and connections between subsystems. There are twc types of subsystems: an executor processing unit for changing process control variables and controller unit for calculating the amounts of the hanges. Figure 5.9 shows the data flow of a feedback close-loop process ontrol system. The connections between the subsystems are the data flow.

A process control system must have the following process control data:

- *Controlled variable:* a target controlled variable such as speed in a cruise control system or the temperature in an auto H/A system. It has a *set point* goal to reach. The controlled variable data should be measured by sensors as a feedback reference to recalculate manipulated variables.
- *Input variable:* a measured input data such as the temperature of return air in a temperature control system.
- *Manipulated variable:* can be adjusted by the controller.

'he input variables and manipulated variables are applied to the execution rocessor which results in a controlled variable. The set point and con-'olled variables are the input data to the controller; the difference between ie controlled variable value and the set point value is used to arrive at a ew manipulated value. Car cruise-control and building temperature con-ol systems are examples of this process control software architecture type f application.


Input variables
Controller
Process
Set point (s)
Manipulated variables (m)
Controlled variable (c)
$\Delta = s - c, m = f(\Delta)$



In a cruise control system there is an engine process unit that drives the wheels and a controller that sets the throttle based on the current wheel speed and set point value. The architecture is similar to the design shown in the preceding diagram.

Applicable domains of process control architecture:

- Embedded software systems involving continuing actions
- Systems that need to maintain an output data at a stable level
- The system can have a set point—the goal the system will reach at its operational level.

Benefits of close-loop feedback process control architecture over open forward architecture:

- It offers a better solution to the control system where no precise formula can be used to decide the manipulated variable.
- The software can be completely embedded in the devices.

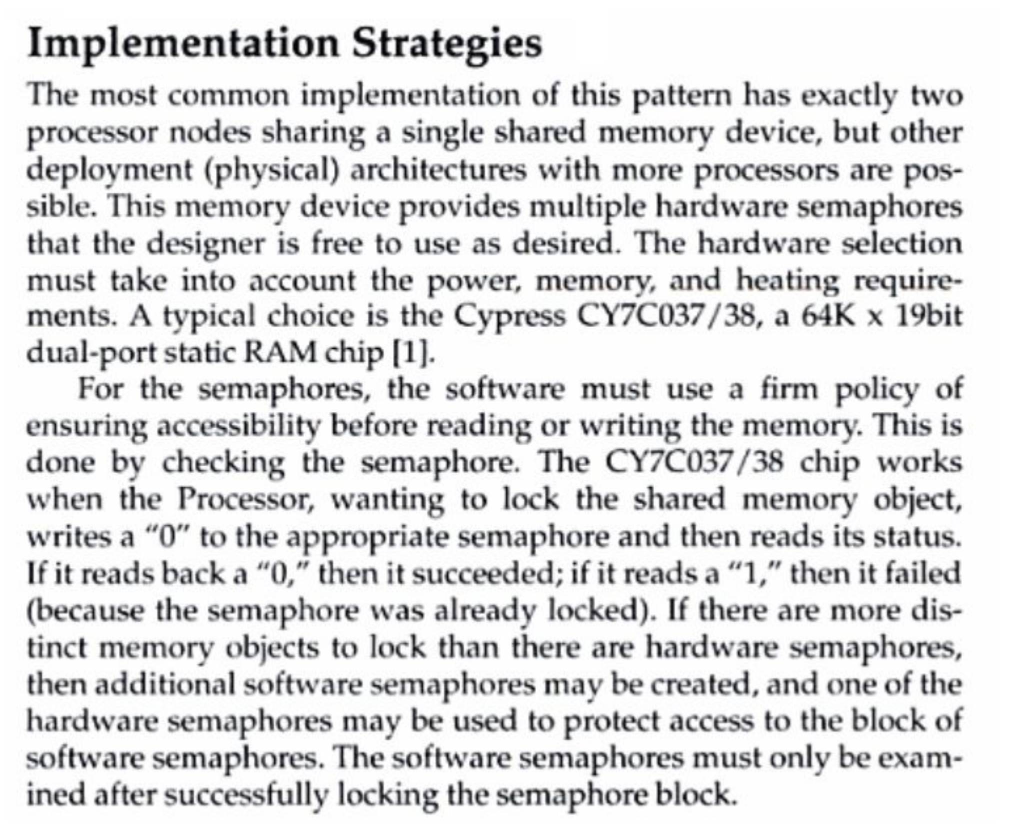
Data-Flow Architecture : *Shared Memory*

Shared Memory is a pattern which can be classified both as
data-flow or real-time architectures

The Shared Memory Pattern uses a common memory area addressable by multiple processors as a means to send messages and share data. This is normally accomplished with the addition of special hardware—specifically, multiported RAM chips.

Problem
Many systems have to share data between multiple processors—this is the essence of distribution, after all. In some cases, the access to the data may persist for a long period of time, and the amount of data shared may be large. In such cases, sending messages may be an inefficient method for sharing such information. Multiple computers may need to update this "global" data, such as in a shared database, or they may need to only read it, as is the case with executable code that may run on many processors or configuration tables. A means by which such data may be effectively shared is needed.

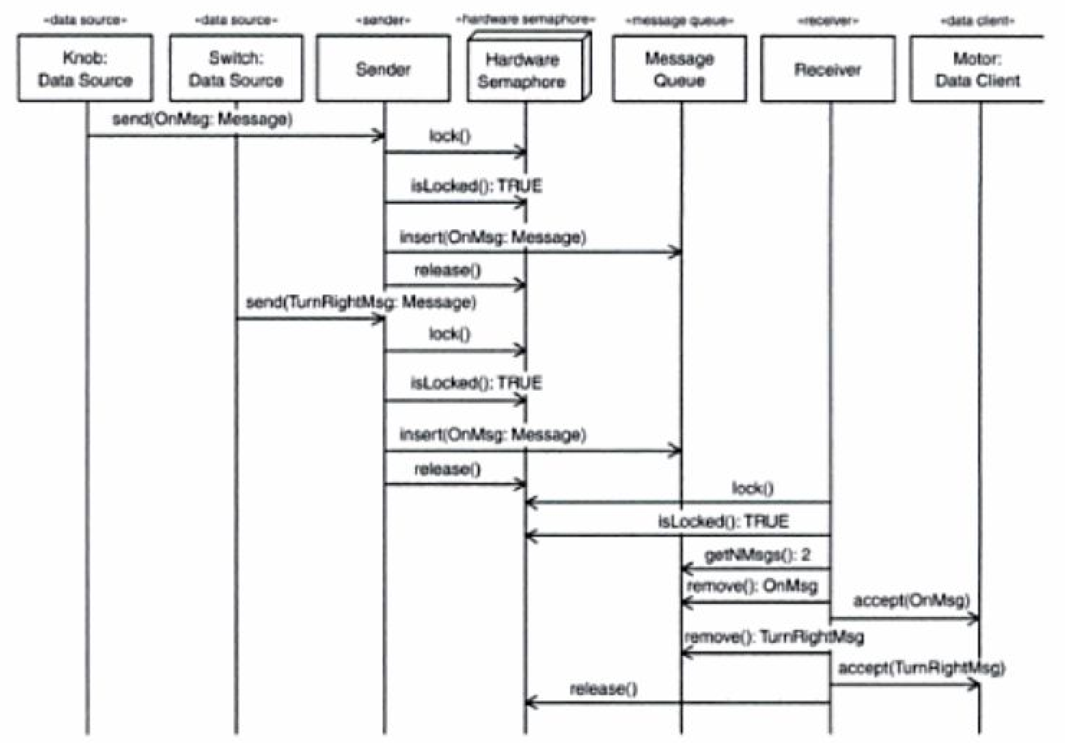
Solution
The Shared Memory Pattern is a simple solution when data must be shared among more than one processor, but timely responses to messages and events between the processors are not required. The pattern almost always involves a combined hardware/software solution. Hardware support for single CPU-cycle semaphore and memory access can avoid memory conflicts and data corruption, but usually some software support to assist the low-level hardware features is required for robust access. If the data to be shared is read-only, as for code that is to be executed on multiple processors, then such concurrency protection mechanisms may not be required.



Implementation Strategies

The most common implementation of this pattern has exactly two processor nodes sharing a single shared memory device, but other deployment (physical) architectures with more processors are possible. This memory device provides multiple hardware semaphores that the designer is free to use as desired. The hardware selection must take into account the power, memory, and heating requirements. A typical choice is the Cypress CY7C037/38, a 64K x 19bit dual-port static RAM chip [1].

For the semaphores, the software must use a firm policy of ensuring accessibility before reading or writing the memory. This is done by checking the semaphore. The CY7C037/38 chip works when the Processor, wanting to lock the shared memory object, writes a "0" to the appropriate semaphore and then reads its status. If it reads back a "0," then it succeeded; if it reads a "1," then it failed (because the semaphore was already locked). If there are more distinct memory objects to lock than there are hardware semaphores, then additional software semaphores may be created, and one of the hardware semaphores may be used to protect access to the block of software semaphores. The software semaphores must only be examined after successfully locking the semaphore block.



6.2 Repository Architecture Style

The repository architecture style is a data-centered architecture that supports user interaction for data processing (as opposed to the batch sequential transaction processing discussed earlier). The software component agents of the data store control the computation and flow of logic of the system. Figure 6.2 gives a general picture of the repository architecture. The dashed lines pointing toward repository in Figure 6.2 indicate that repository clients have full control over the logic flow. Clients can get data from the data store and put data in the data store. Different clients may have different interfaces and different data access privileges.

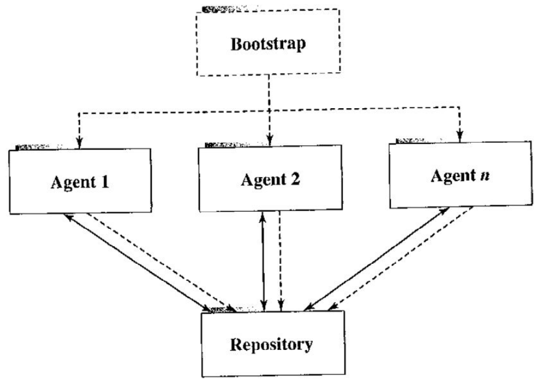
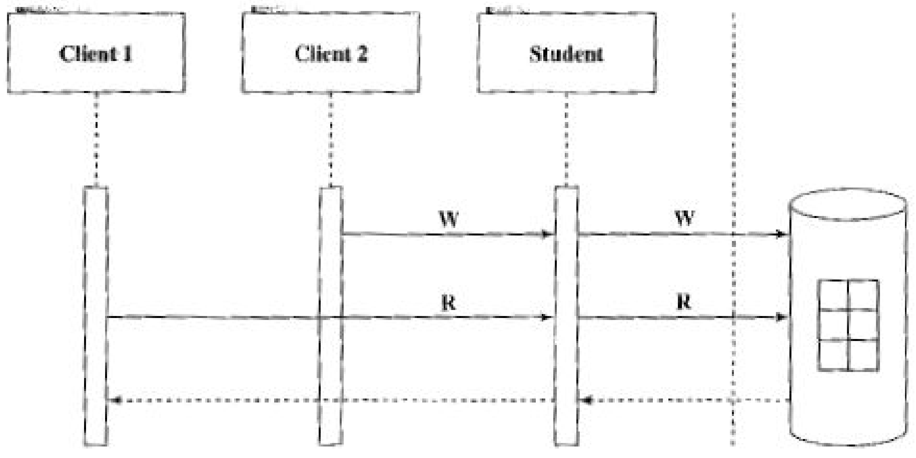
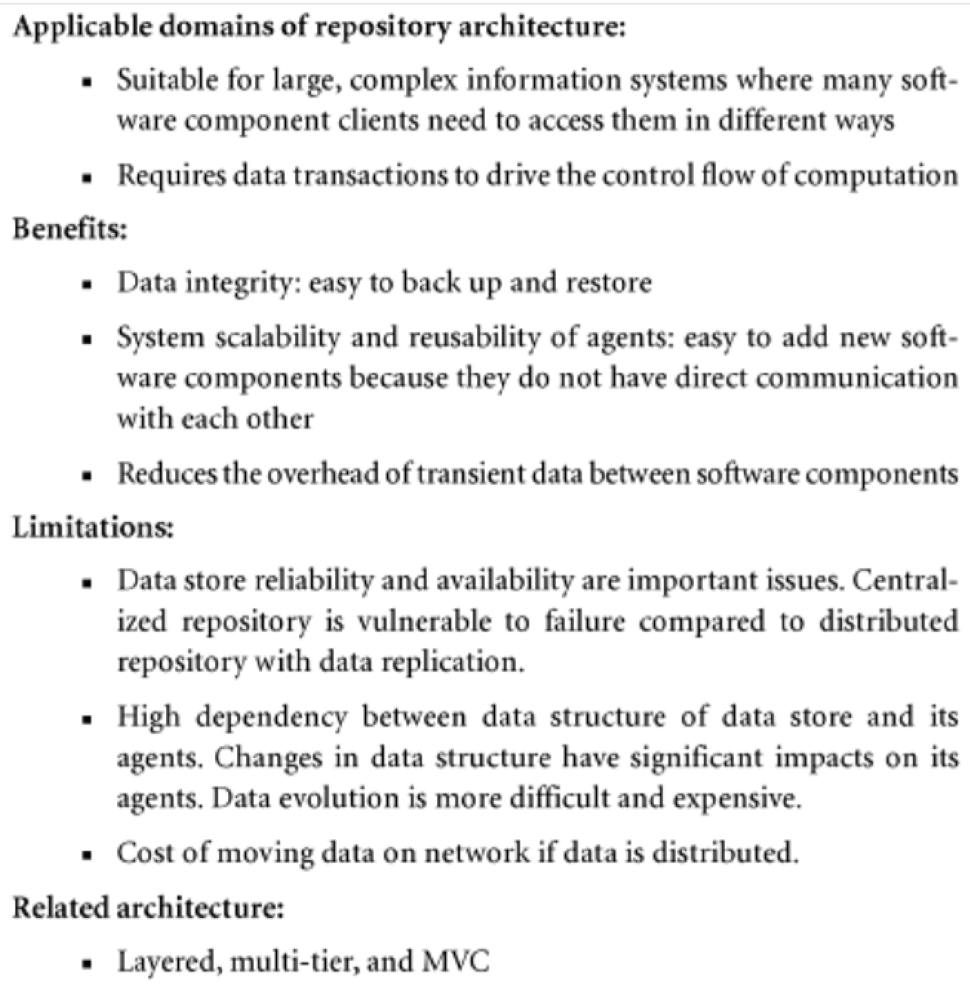


Figure 6.2
Repository architecture

The relational database management system is a typical design domain for repositoty architecture.





Applicable domains of repository architecture:

- Suitable for large, complex information systems where many software component clients need to access them in different ways
- Requires data transactions to drive the control flow of computation

Benefits:

- Data integrity: easy to back up and restore
- System scalability and reusability of agents: easy to add new software components because they do not have direct communication with each other
- Reduces the overhead of transient data between software components

Limitations:

- Data store reliability and availability are important issues. Centralized repository is vulnerable to failure compared to distributed repository with data replication.
- High dependency between data structure of data store and its agents. Changes in data structure have significant impacts on its agents. Data evolution is more difficult and expensive.
- Cost of moving data on network if data is distributed.

Related architecture:

- Layered, multi-tier, and MVC

Data-centered Software Architecture *Blackboard*

The *Blackboard* architectural pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Context An immature domain in which no closed approach to a solution is known or feasible.

Problem The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures, such as diagrams, tables or English phrases. Vision, image recognition, speech recognition and surveillance are examples of domains in which such problems occur. They are characterized by a problem that, when decomposed into subproblems, spans several fields of expertise. The solutions to the partial problems require different representations and paradigms. In many cases no predetermined strategy exists for how the 'partial problem solvers' should combine their knowledge. This is in contrast to functional decomposition, in which several solution steps are arranged so that the sequence of their activation is hard-coded.

In some of the above problem domains you may also have to work with uncertain or approximate knowledge. Each transformation step can also generate several alternative solutions. In such cases it is often enough to find an optimal solution for most cases, and a suboptimal solution, or no solution, for the rest. The limitations of a Blackboard system therefore have to be documented carefully, and if important decisions depend on its results, the results have to be verified.

Solution The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure. Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution. These specialized programs are independent of each other. They do not call each other, nor is there a predetermined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of progress. A central control component evaluates the current state of processing and coordinates the specialized programs. This data-directed control regime is referred to as *opportunistic problem solving*. It makes experimentation with different algorithms possible, and allows experimentally-derived heuristics to control processing.

During the problem-solving process the system works with partial solutions that are combined, changed or rejected. Each of these solutions represents a partial problem and a certain stage of its solution. The set of all possible solutions is called the *solution space*, and is organized into levels of abstraction. The lowest level of solution consists of an internal representation of the input. Potential solutions of the overall system task are on the highest level.

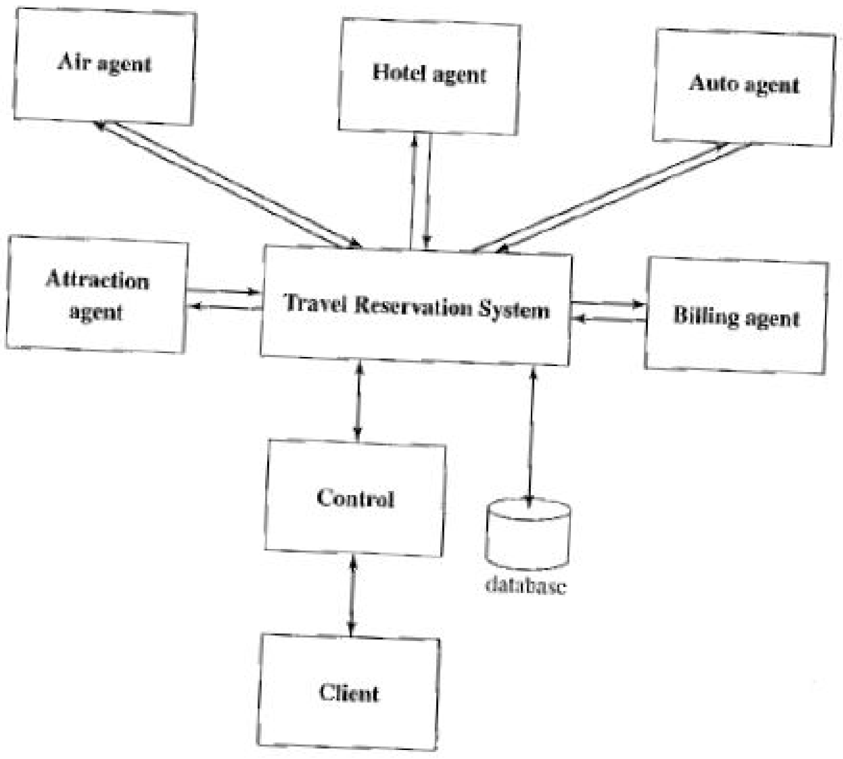


Figure 6.12
Blackboard architecture for a travel consulting system

Benefits:

- *Scalability:* easy to add or update knowledge source.
- *Concurrency:* all knowledge sources can work in parallel since they are independent of each other.
- Supports experimentation for hypotheses.
- Reusability of knowledge source agents.

Limitations:

- Due to the close dependency between the blackboard and knowledge source, the structure change of the blackboard may have a significant impact on all of its agents.
- Since only partial or approximate solutions are expected, it can be difficult to decide when to terminate reasoning.
- Synchronization of multiple agents is an issue. Since multiple agents are working and updating the shared data in the blackboard simultaneously, the preference or priority of executions of multiple agents must be coordinated.
- Debugging and testing of the system is a challenge.

Related architecture:

- Implicit invocation architecture such as event-based, MVC architecture

7.3 Master-Slave

The master-slave architecture is a variant of the main-subroutine architecture style that supports fault tolerance and system reliability. In this architecture, slaves provide replicated services to the master, and the master selects a particular result among slaves by certain selection strategies. The slaves may perform the same functional task by different algorithms and methods or by a totally different functionality.

Figure 7.4 shows a master-slave architecture where all slaves implement the same service. The master configures the invocations of the replicated services and receives the results back from all slaves. It then determines which of the returned results will be selected.

The diagram in Figure 7.5 shows a UML class diagram for the master-slave architecture where multiple slave classes implement the same interface in different ways.

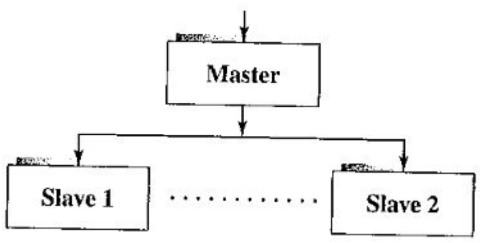


Figure 7.4
Block diagram for master-slave architecture

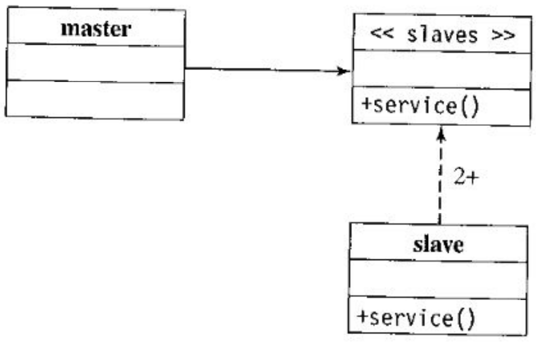
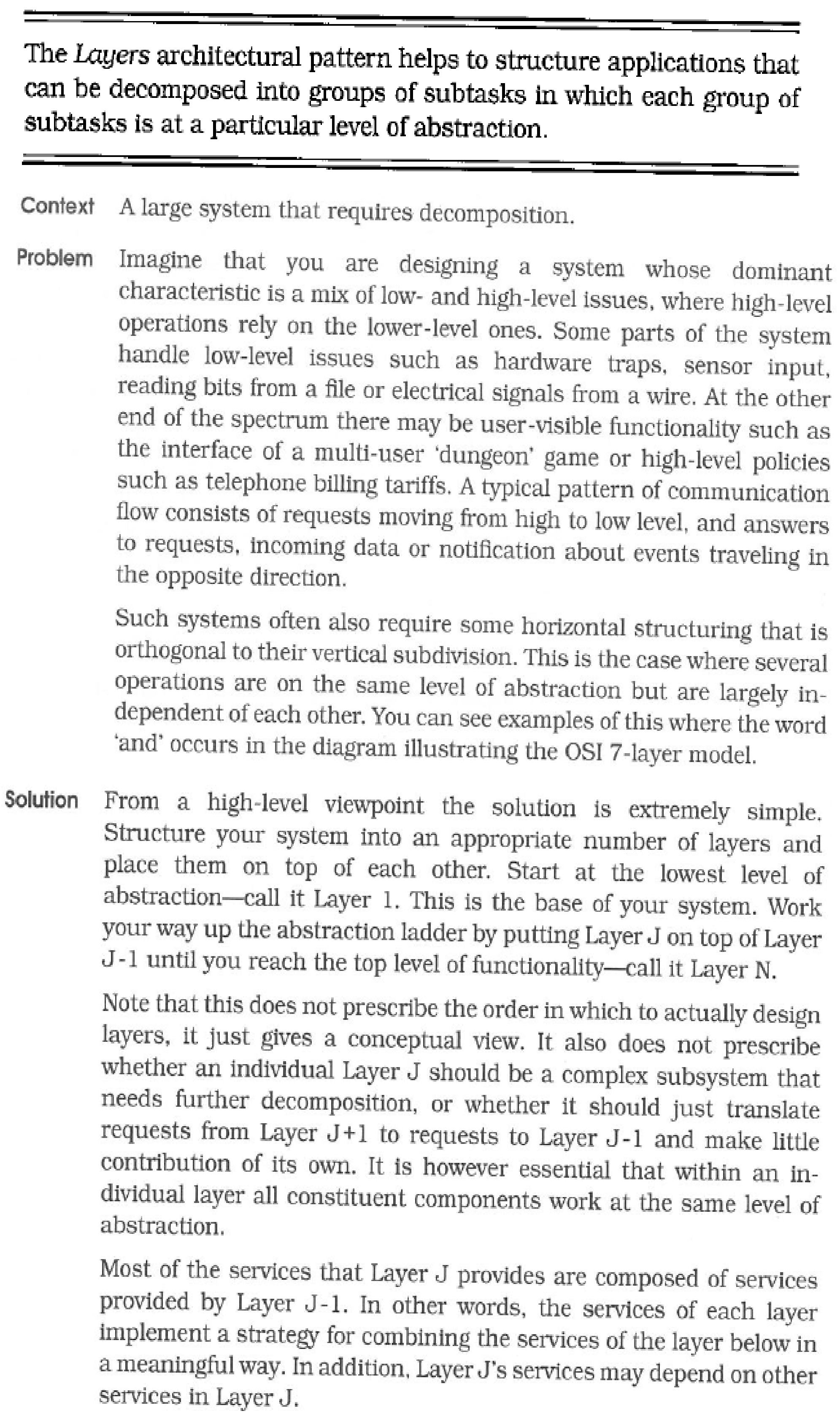


Figure 7.5
Class diagram for master-slave architecture

Hierarchical Architecture *Layered*



The *Layers* architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Context A large system that requires decomposition.

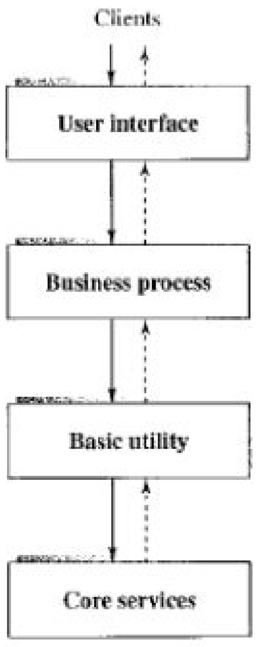
Problem Imagine that you are designing a system whose dominant characteristic is a mix of low- and high-level issues, where high-level operations rely on the lower-level ones. Some parts of the system handle low-level issues such as hardware traps, sensor input, reading bits from a file or electrical signals from a wire. At the other end of the spectrum there may be user-visible functionality such as the interface of a multi-user 'dungeon' game or high-level policies such as telephone billing tariffs. A typical pattern of communication flow consists of requests moving from high to low level, and answers to requests, incoming data or notification about events traveling in the opposite direction.

Such systems often also require some horizontal structuring that is orthogonal to their vertical subdivision. This is the case where several operations are on the same level of abstraction but are largely independent of each other. You can see examples of this where the word 'and' occurs in the diagram illustrating the OSI 7-layer model.

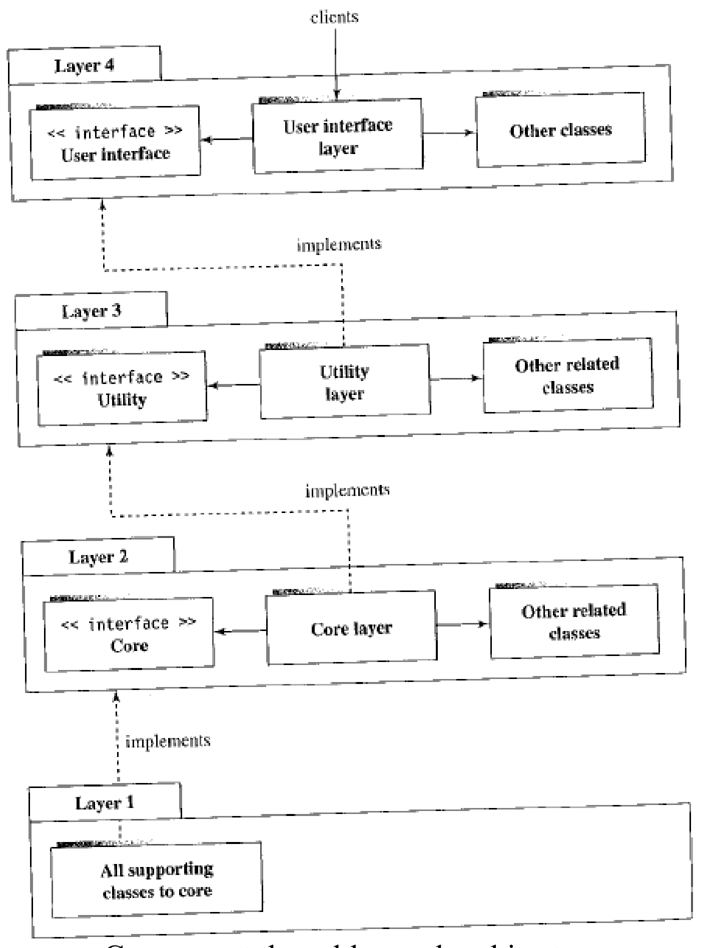
Solution From a high-level viewpoint the solution is extremely simple. Structure your system into an appropriate number of layers and place them on top of each other. Start at the lowest level of abstraction—call it Layer 1. This is the base of your system. Work your way up the abstraction ladder by putting Layer J on top of Layer J-1 until you reach the top level of functionality—call it Layer N.

Note that this does not prescribe the order in which to actually design layers, it just gives a conceptual view. It also does not prescribe whether an individual Layer J should be a complex subsystem that needs further decomposition, or whether it should just translate requests from Layer J+1 to requests to Layer J-1 and make little contribution of its own. It is however essential that within an individual layer all constituent components work at the same level of abstraction.

Most of the services that Layer J provides are composed of services provided by Layer J-1. In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, Layer J's services may depend on other services in Layer J.



A partial layered-architecture:
A business example



Component_based layered architecture

Benefits:

- Incremental software development based on increasing levels of abstraction.
- Enhanced independence of upper layer to lower layer since there is no impact from the changes of lower layer services as long as their interfaces remain unchanged.
- Enhanced flexibility: interchangeability and reusability are enhanced due to the separation of the standard interface and its implementation.
- Component-based technology is a suitable technology to implement layered architecture; this makes it much easier for the system to allow for plug-and-play of new components.
- Promotion of portability: each layer can be an abstract machine (see Section 7.5) deployed independently.

Limitations:

- Lower runtime performance since a client's request or a response to a client must go through potentially several layers. There are also performance concerns of overhead on the data marshaling and buffering by each layer.
- Many applications cannot fit this architecture design.
- Breach of interlayer communication may cause deadlocks, and "bridging" may cause tight coupling.
- Exceptions and error handling are issues in the layered architecture, since faults in one layer must propagate upward to all calling layers.

Distributed Architecture *Client-Server*

The client/server architectural style describes distributed systems that involve a separate client and server system, and a connecting network. The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style.

Historically, client/server architecture indicated a graphical desktop UI application that communicated with a database server containing much of the business logic in the form of stored procedures, or with a dedicated file server. More generally, however, the client/server architectural style describes the relationship between a client and one or more servers, where the client initiates one or more requests (perhaps using a graphical UI), waits for replies, and processes the replies on receipt. The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client.

Today, some examples of the client/server architectural style include Web browser—based programs running on the Internet or an intranet; Microsoft Windows® operating system—based applications that access networked data services; applications that access remote data stores (such as e-mail readers, FTP clients, and database query tools); and tools and utilities that manipulate remote systems (such as system management tools and network monitoring tools).

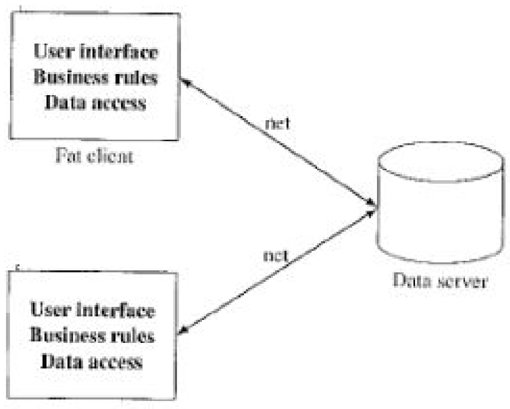


Figure 10.1
Two-tier client-server architecture

The main benefits of the client/server architectural style are:

- **Higher security**. All data is stored on the server, which generally offers a greater control of security than client machines.
- **Centralized data access**. Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- **Ease of maintenance**. Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

Consider the client/server architectural style if your application is server based and will support many clients, you are creating Web-based applications exposed through a Web browser, you are implementing business processes that will be used by people throughout the organization, or you are creating services for other applications to consume. The client/server architectural style is also suitable, like many networked styles, when you want to centralize data storage, backup, and management functions, or when your application must support different client types and different devices.

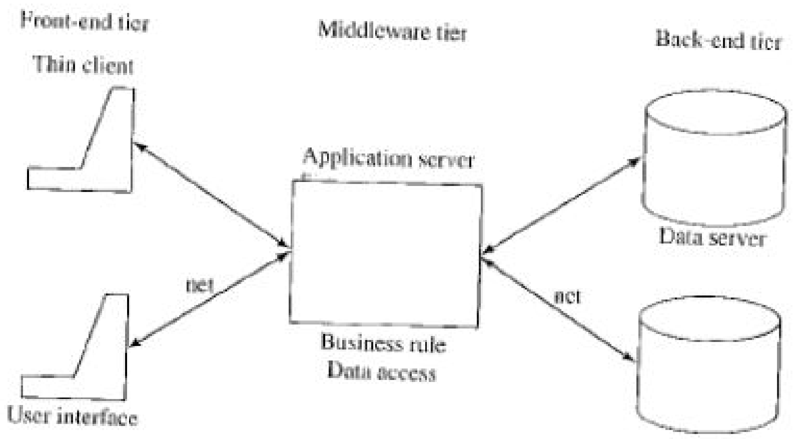
However, the traditional 2-Tier client/server architectural style has numerous disadvantages, including the tendency for application data and business logic to be closely combined on the server, which can negatively impact system extensibility and scalability, and its dependence on a central server, which can negatively impact system reliability. To address these issues, the client-server architectural style has evolved into the more general 3-Tier (or N-Tier) architectural style, described below, which overcomes some of the disadvantages inherent in the 2-Tier client-server architecture and provides additional benefits.

Distributed Architecture 3-tiers & *Multi-tiers*

N-tier and 3-tier are architectural deployment styles that describe the separation of functionality into segments in much the same way as the layered style, but with each segment being a tier that can be located on a physically separate computer. They evolved through the component-oriented approach, generally using platform specific methods for communication instead of a message-based approach.

N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization. Each tier is completely independent from all other tiers, except for those immediately above and below it. The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th tier (if there is one), and how to handle the results of the request. Communication between tiers is typically asynchronous in order to support better scalability.

N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality. When using a layered design approach, a layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.



Three-tier architecture

An example of the N-tier/3-tier architectural style is a typical financial Web application where security is important. The business layer must be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter network. Another example is a typical rich client connected application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on one or more server tiers.

The main benefits of the N-tier/3-tier architectural style are:

- **Maintainability**. Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- **Scalability**. Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- **Flexibility**. Because each tier can be managed or scaled independently, flexibility is increased.
- **Availability**. Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Consider either the N-tier or the 3-tier architectural style if the processing requirements of the layers in the application differ such that processing in one layer could absorb sufficient resources to slow the processing in other layers, or if the security requirements of the layers in the application differ. For example, the presentation layer should not store sensitive data, while this may be stored in the business and data layers. The N-tier or the 3-tier architectural style is also appropriate if you want to be able to share business logic between applications, and you have sufficient hardware to allocate the required number of servers to each tier.

Consider using just three tiers if you are developing an intranet application where all servers are located within the private network; or an Internet application where security requirements do not restrict the deployment of business logic on the public facing Web or application server. Consider using more than three tiers if security requirements dictate that business logic cannot be deployed to the perimeter network, or the application makes heavy use of resources and you want to offload that functionality to another server

Distributed Architecture *Broker*

The *Broker* architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

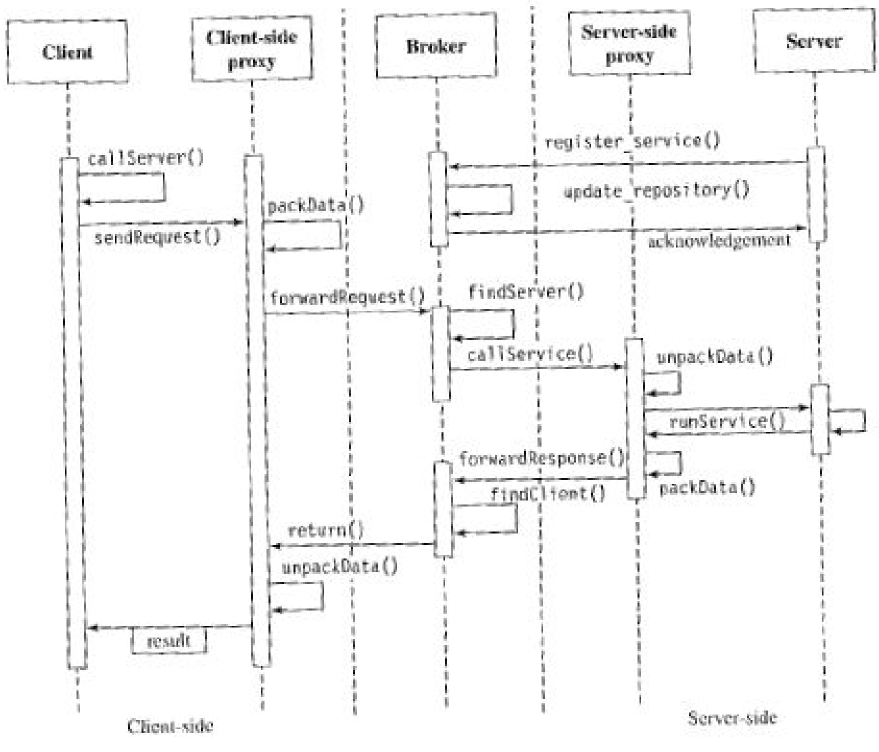
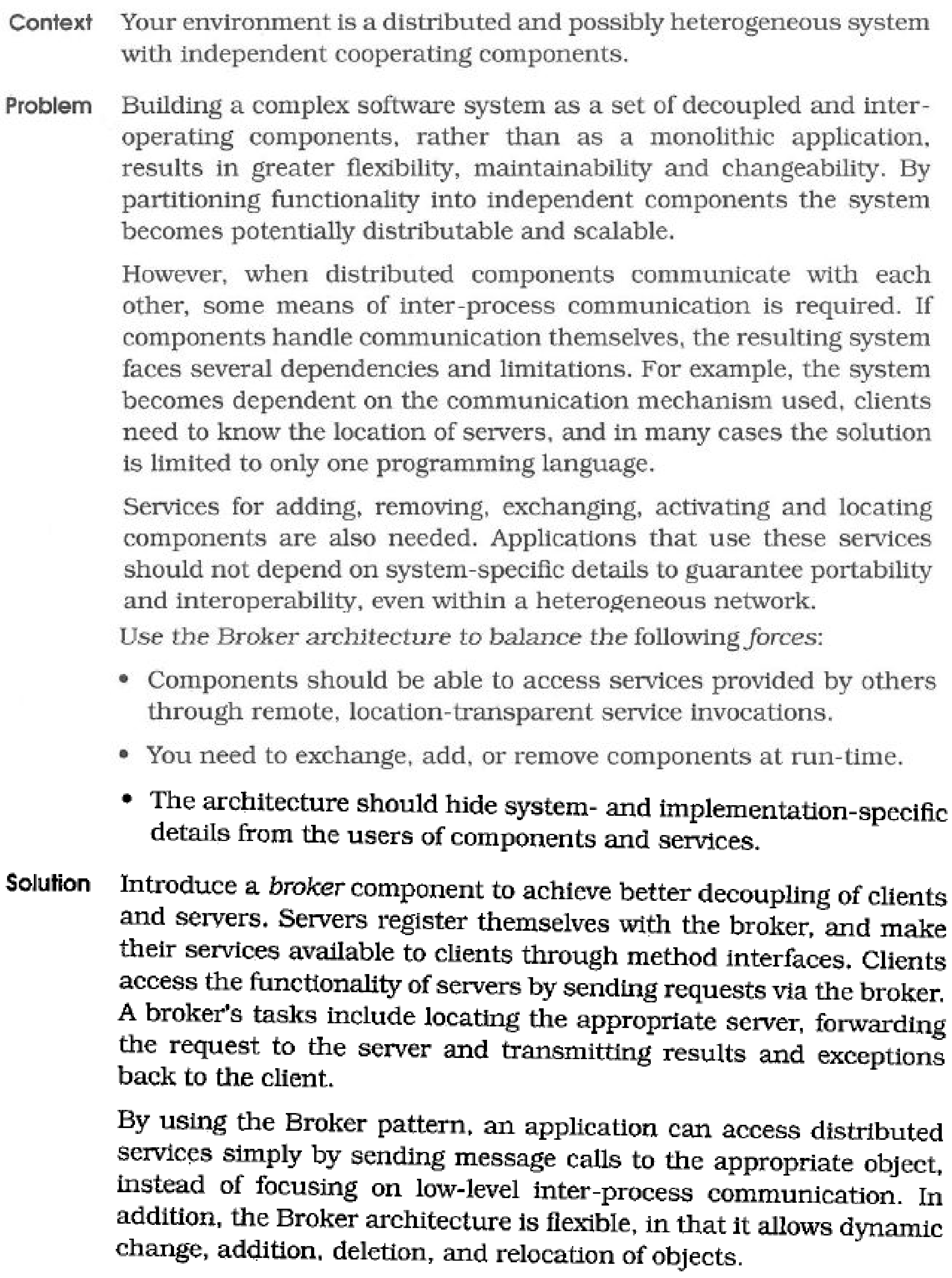


Figure 10.6
Sequence diagram for broker architecture



Context Your environment is a distributed and possibly heterogeneous system with independent cooperating components.

Problem Building a complex software system as a set of decoupled and inter-operating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentially distributable and scalable.

However, when distributed components communicate with each other, some means of inter-process communication is required. If components handle communication themselves, the resulting system faces several dependencies and limitations. For example, the system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language.

Services for adding, removing, exchanging, activating and locating components are also needed. Applications that use these services should not depend on system-specific details to guarantee portability and interoperability, even within a heterogeneous network.

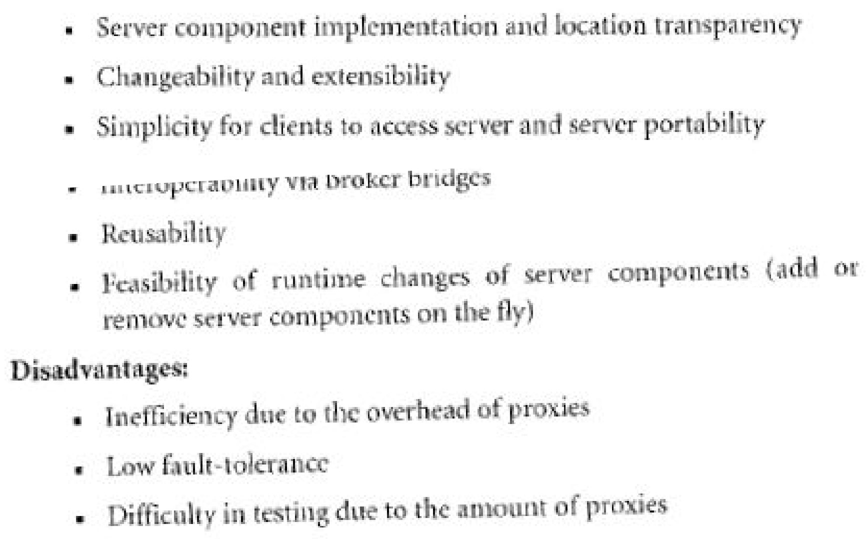
Use the Broker architecture to balance the following *forces*:

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

Solution Introduce a *broker* component to achieve better decoupling of clients and servers. Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

By using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication. In addition, the Broker architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects.

Avantages:

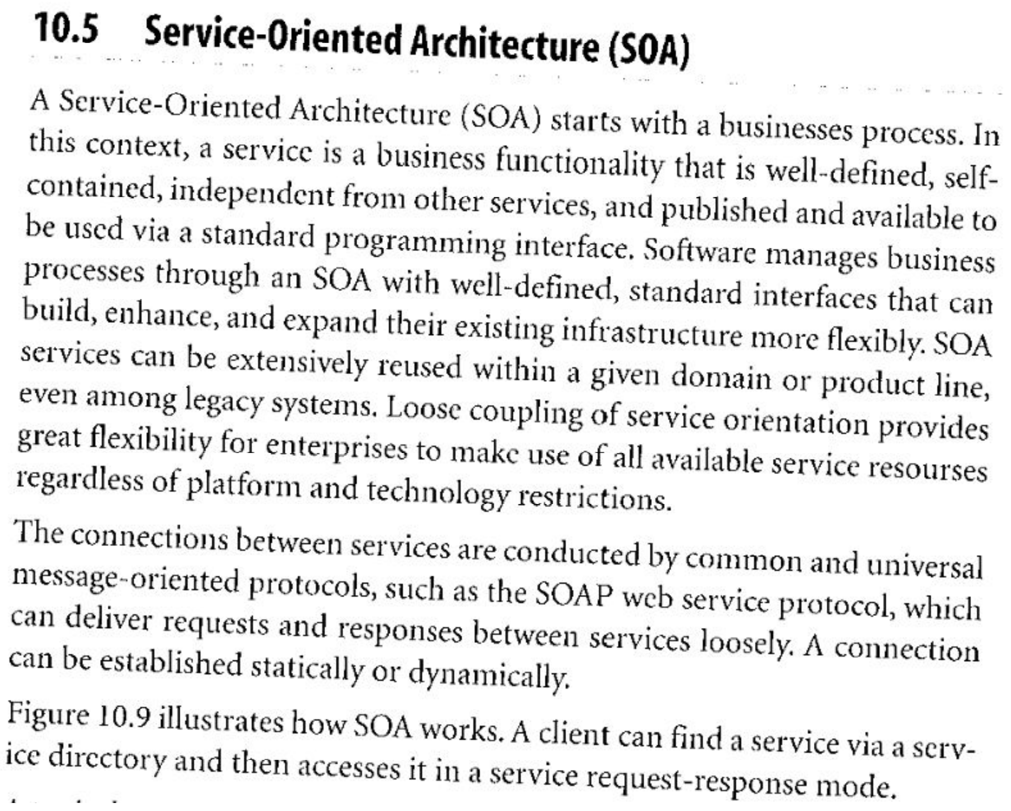


- Server component implementation and location transparency
- Changeability and extensibility
- Simplicity for clients to access server and server portability
- Interoperability via broker bridges
- Reusability
- Feasibility of runtime changes of server components (add or remove server components on the fly)

Disadvantages:

- Inefficiency due to the overhead of proxies
- Low fault-tolerance
- Difficulty in testing due to the amount of proxies

Distributed Architecture *Service Oriented Architecture*



10.5 Service-Oriented Architecture (SOA)

A Service-Oriented Architecture (SOA) starts with a businesses process. In this context, a service is a business functionality that is well-defined, self-contained, independent from other services, and published and available to be used via a standard programming interface. Software manages business processes through an SOA with well-defined, standard interfaces that can build, enhance, and expand their existing infrastructure more flexibly. SOA services can be extensively reused within a given domain or product line, even among legacy systems. Loose coupling of service orientation provides great flexibility for enterprises to make use of all available service resourses regardless of platform and technology restrictions.

The connections between services are conducted by common and universal message-oriented protocols, such as the SOAP web service protocol, which can deliver requests and responses between services loosely. A connection can be established statically or dynamically.

Figure 10.9 illustrates how SOA works. A client can find a service via a service directory and then accesses it in a service request-response mode.

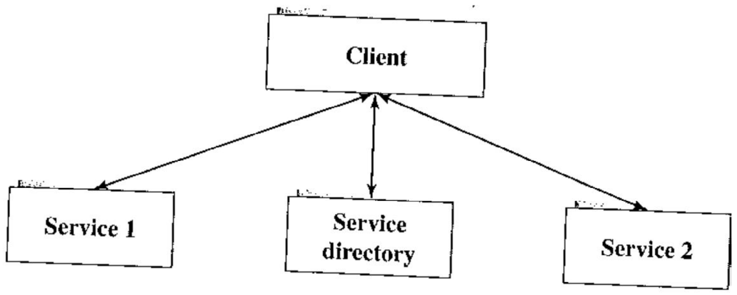


Figure 10.9
Client with services and service directory

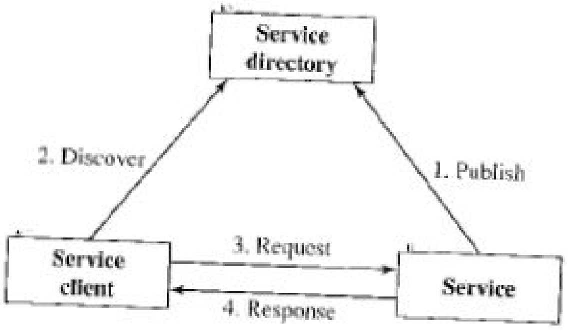


Figure 10.13
Service working model

Advantages of SOA:

- *Loosely-coupled connections:* Loose-coupling is the key attribute of service-oriented architecture. Each service component is independent due to the stateless service feature. The implementation of a service will not affect its application as long as the exposed interface is not changed. This makes SOA software much easier to evolve and update.
- *Interoperability:* Technically, any client or service can access other services regardless of their platform, technology, vendors, or language implementations.
- *Reusability:* Any service can be reused by any other service. Because clients of a service need only to know its public interfaces, service composition and integration become much easier. This makes

SOA-based business application development much more efficient in terms of time and cost.

- *Scalability:* Loosely-coupled services are easy to scale. The coarse-grained, document-oriented, and asynchronous service features enhance the scalability attribute.

Interaction-Oriented Software Architecture *MVC*

The *Model-View-Controller* architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

Context Interactive applications with a flexible human-computer interface.

Problem User interfaces are especially prone to change requests. When you extend the functionality of an application, you must modify menus to access these new functions. A customer may call for a specific user interface adaptation, or a system may need to be ported to another platform with a different 'look and feel' standard. Even upgrading to a new release of your windowing system can imply code changes. The user interface platform of long-lived systems thus represents a moving target.

Different users place conflicting requirements on the user interface. A typist enters information into forms via the keyboard. A manager wants to use the same system mainly by clicking icons and buttons. Consequently, support for several user interface paradigms should be easily incorporated.

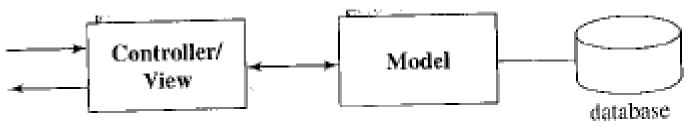
Solution Model-View-Controller (MVC) was first introduced in the Smalltalk-80 programming environment [KP88]. MVC divides an interactive application into the three areas: *processing*, *output*, and *input*.

The *model* component encapsulates core data and functionality. The model is independent of specific output representations or input behavior.

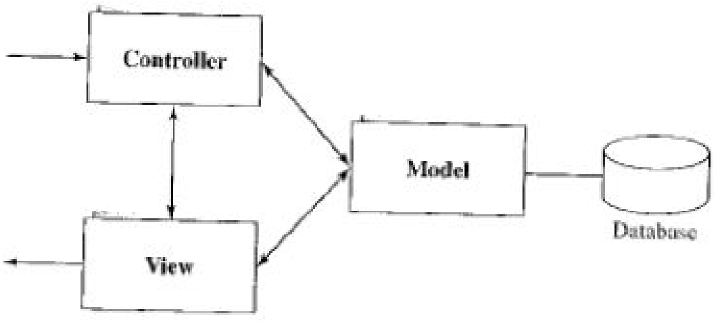
View components display information to the user. A view obtains the data from the model. There can be multiple views of the model.

Each view has an associated *controller* component. Controllers receive input, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input. Events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.

The separation of the model from view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all other views dependent on this data should reflect the changes. The model therefore notifies all views whenever its data changes. The views in turn retrieve new data from the model and update the displayed information. This change-propagation mechanism is described in the Publisher-Subscriber pattern (339).



MVC-I



MVC-II

Benefits:

- Many MVC vendor framework toolkits are available.
- Multiple views synchronized with same data model.
- Easy to change or plug in new interface views, allowing updating of interface views with new technologies without overhauling the rest of the system.
- Very effective for developments if graphics, programming, and database development professionals are working in a team in a designed project.

Limitations:

- Not suitable for agent-oriented applications such as interactive mobile and robotics applications.
- Multiple pairs of controllers and views based on the same data model make any data model change expensive.
- The division between the View and the Controller is not clear in some cases.

Related architecture:

- PAC, and implicit invocation such as event-based, multi-tier architecture

Interaction-Oriented Software Architecture *PAC*

The *Presentation-Abstraction-Control* architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Problem Interactive systems can often be viewed as a set of cooperating agents. Agents specialized in human-computer interaction accept user input and display data. Other agents maintain the data model of the system and offer functionality that operates on this data. Additional agents are responsible for diverse tasks such as error handling or communication with other software systems. Besides this horizontal decomposition of system functionality, we often encounter a vertical decomposition. Production planning systems (PPS), for example, distinguish between production planning and the execution of a previously specified production plan. For each of these tasks separate agents can be defined.

In such an architecture of cooperating agents, each agent is specialized for a specific task, and all agents together provide the system functionality. This architecture also captures both a horizontal and vertical decomposition.

Solution Structure the interactive application as a tree-like hierarchy of *PAC agents*. There should be one top-level agent, several intermediate-level agents, and even more bottom-level agents. Every agent is responsible for a specific aspect of the application's functionality, and consists of three components: presentation, abstraction, and control.

The whole hierarchy reflects transitive dependencies between agents. Each agent depends on all higher-level agents up the hierarchy to the top-level agent.

The agent's *presentation* component provides the visible behavior of the PAC agent. Its *abstraction* component maintains the data model that underlies the agent, and provides functionality that operates on this data. Its *control* component connects the presentation and abstraction components, and provides functionality that allows the agent to communicate with other PAC agents.

The *top-level PAC agent* provides the functional core of the system. Most other PAC agents depend or operate on this core. Furthermore, the top-level PAC agent includes those parts of the user interface that cannot be assigned to particular subtasks, such as menu bars or a dialog box displaying information about the application.

Bottom-level PAC agents represent self-contained semantic concepts on which users of the system can act, such as spreadsheets and charts. The bottom-level agents present these concepts to the user and support all operations that users can perform on these agents, such as zooming or moving a chart.

Intermediate-level PAC agents represent either combinations of, or relationships between, lower-level agents. For example, an intermediate-level agent may maintain several views of the same data, such as a floor plan and an external view of a house in a CAD system for architecture.

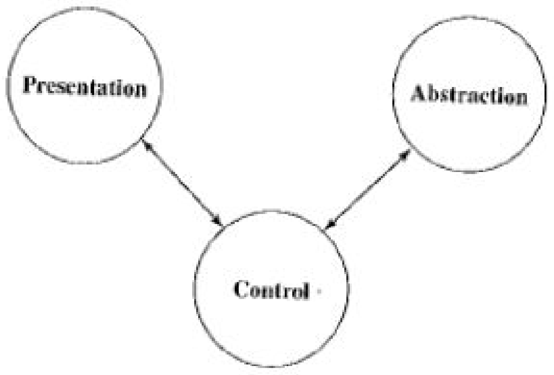
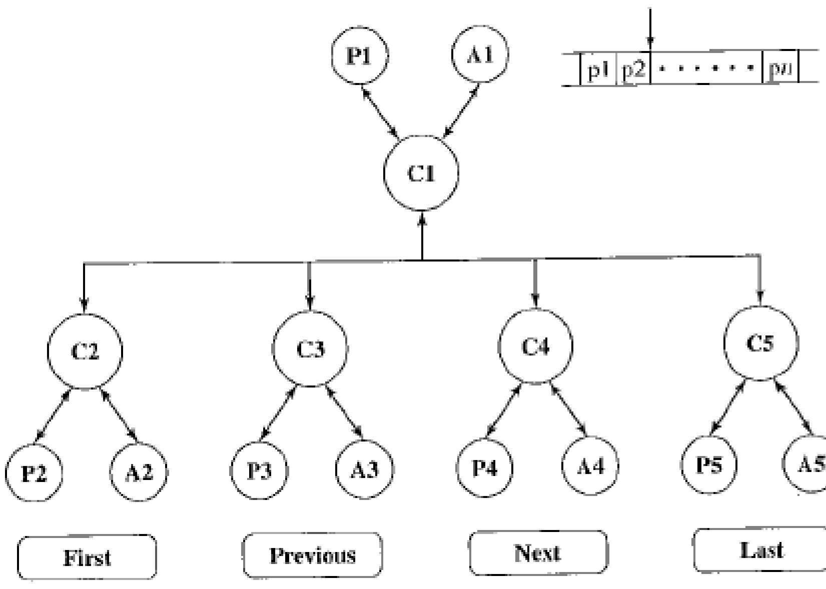
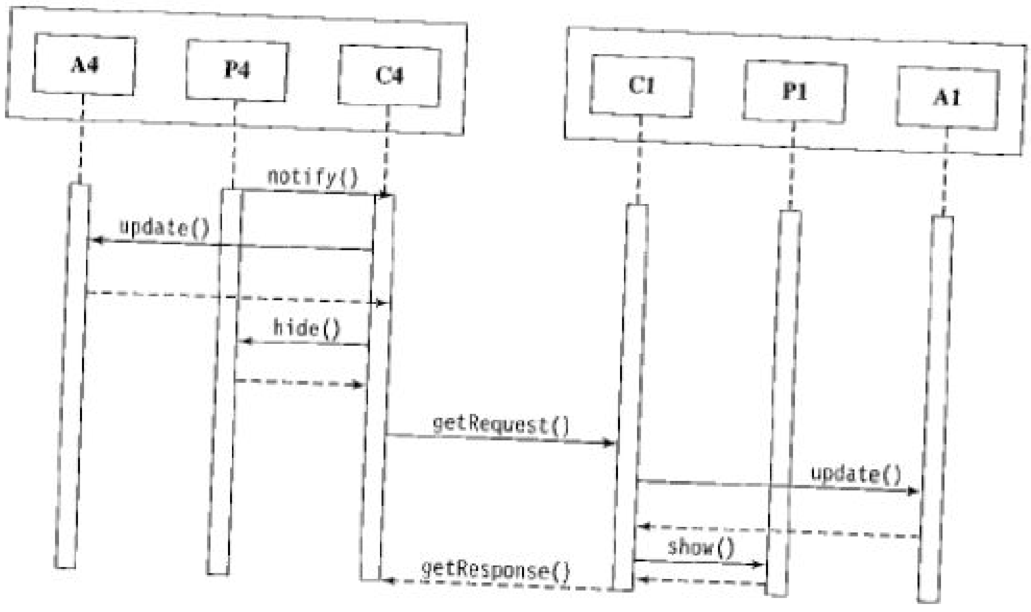


Figure 9.6
A single agent in PAC



Multiple-agents in PAC



Applicable domains of the PAC architecture:

- Suitable for an interactive system where the system can be divided into many cooperating agents in a hierarchical manner and each agent has its own specific assigned job.
- Suitable when the coupling among the agents is expected to be loose so that changes on an agent do not affect others.

Benefits:

- Support of multitasking and multiviewing
- Support agent reusability and extensibility
- Easy to plug in new agent or replace an existing one
- Support for concurrency where multiple agents run in parallel in different threads or on different devices or computers

Limitations:

- Extra time lost due to the control bridge between presentation and abstraction and the communication of controls among agents.
- Difficult to determine the correct number of the agents due to the loose coupling and high independence between agents.

Sommaire

Data-Flow architecture

- Pipe and filter
- Process control architecture
- Shared-memory

Data-centered architecture

- Repository architecture
- Blackboard

Hierarchical

- Master-slave
- Layered

Distributed-architecture

- Client-server
- 3-tiers & multi-tiers
- Broker
- SOA

Interaction-oriented architecture

- MVC
- PAC

Bibliographie

- **Pattern-Oriented Software Architecture**, Volume 1: A System of Patterns, de F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Wiley
- **Microsoft® Application Architecture Guide**, de Microsoft patterns & practices. 2009.
- **Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems**, de B. Powel Douglass
- **Software architecture and Design Illuminated**, de K. Qian, X. Fu, L. Tao, C.-W. Xu, J. L. Dias Herrera, John and Barret Publishers. 2010