PART B

UNIT 5

Distributed Systems – Broker Architecture

There are two major trends in recent developments in hardware technology:

- Computer systems with multiple CPUs are entering even small offices, notably multiprocessing systems running operating systems such as IBM OS/2 Warp, Microsoft Windows NT, or UNIX
- Local area networks connecting hundreds of heterogeneous computers have become commonplace.

Advantages

- Economics – better price/performance ratio
- Performance and Scalability – The n/w is the computer
- Inherent distribution
- Reliability

Drawbacks:

- Distributed systems need radically different software than do centralized systems
  - OMG and companies such as Microsoft developed their own technologies

Types of Patterns

- The Pipes and Filters pattern
  - Provides a structure for systems that process a stream of data.
  - Processing Step is encapsulated in filters and data is passes via pipes between adjacent filters
- The Microkernel pattern
  - Applies to software systems that must be able to adapt to changing system requirements.
  - Separates a minimal functional core from extended functionality and customer-specific parts.

- The Broker 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.
  - Three groups of users are benefited
    - Already working with a broker system and need to know architecture
    - Those who wish to build the lean version of the broker
    - Those who want to build full fledged broker system

Broker – Definition

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.

Example:

Suppose we are developing a city information system (CIS) designed to run on a wide area network. Some computers in the network host one or more services that maintain information about events, restaurants, hotels, historical monuments or public transportation. Computer terminals are connected to the network. Tourists throughout the city can retrieve information in which they are interested from the terminals using a World Wide Web IWWW) browser. This front-end software supports the on-line retrieval of information from the appropriate servers and its display on the screen. The data is distributed across the network and is not **all** maintained in the terminals.

- We expect the system to change and grow continuously, so the individual services should be decoupled from each other.
- The terminal software should be able to access services without having to know their location.
- One solution is to install a separate network that connects all terminals and servers – Intranet
- Disadvantages
  - Not every Information provider wants to connect to a closed intranet
  - Available services should be accessed from all over the world

Context: Environment is a distributed and possibly heterogeneous system with independent cooperating components.

Problem: It is a good idea to build a system as a set of decoupled interoperating components. However, some means of inter-process communication is required. If the components handle this by themselves, leads to dependencies and limitations. Example, system depending on communicating mechanism used, clients need to know the location of server and limitation of solution in one prog. 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
- From a developer's viewpoint, there should essentially be no difference between developing software for centralized systems and developing for distributed ones.
- An application that uses an object should only see the interface offered by the object. It should not need to know anything about the implementation details of an object, or about its physical location.

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.
- In addition, the Broker architecture is flexible… how?
- The Broker pattern reduces the complexity involved in developing distributed applications.
- Broker systems therefore offer a path to the integration of two core technologies: distribution and object technology.
- They also extend object models from single applications to distributed applications.

The Broker architectural pattern comprises six types of participating components: ***clients, servers, brokers, bridges, client-side proxies*** and ***server-side proxies.***

- A ***server*** implements objects that expose their functionality through interfaces that consist of operations and attributes.
- These interfaces are made available either through an interface definition language (IDL) or through a binary standard.
- There are two kinds of servers
  - Offering common services
  - Implementing Specific functionality

Server in our CIS example –

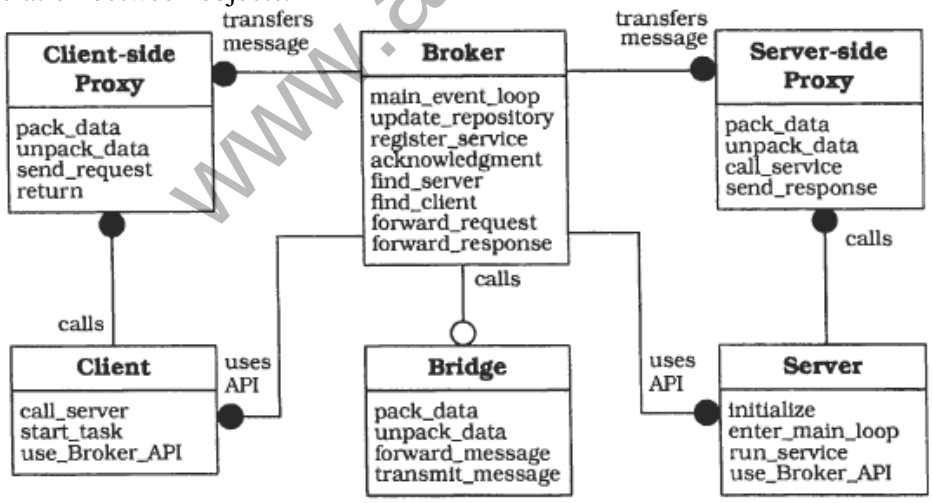
- WWW servers that provide access to HTML pages
- Implemented as httpd daemon processes that wait on specific ports for incoming requests.
- When a request arrives at the server, the requested document and any additional data is sent to the client using data streams.

Additionally, there exist CGI Scripts, Applets, and PHP etc

- ***Clients*** are applications that access the services of at least one server.

- To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.
- Clients do not need to know the location of the servers they access.
- It allows the addition of new services and the movement of existing services to other locations, even while the system is running.
- In the context of the Broker pattern, the clients are the available **WWW browsers.**
- A ***broker*** is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client.
- A broker must have some means of locating the receiver of a request based on its unique system identifier.
- A broker offers APIs to clients and servers that include operations for registering servers and for invoking server methods.
- A broker in our CIS example is the combination of an Internet gateway and the Internet infrastructure itself.
- ***Client-side proxies*** represent a layer between clients and the broker.
- This additional layer provides transparency, in that a remote object appears to the client as a local one.
- In many cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client.
- ***Server-side proxies*** are generally analogous to Client-side proxies
- The difference is that they are responsible for receiving requests, unpacking incoming messages, unmarshaling the parameters, and calling the appropriate service.
- ***Bridges*** are optional components used for hiding implementation details when two brokers interoperate.
- A bridge builds a layer that encapsulates all the system-specific details.
- Bridges are not necessary in CIS because they all use http/ftp in common

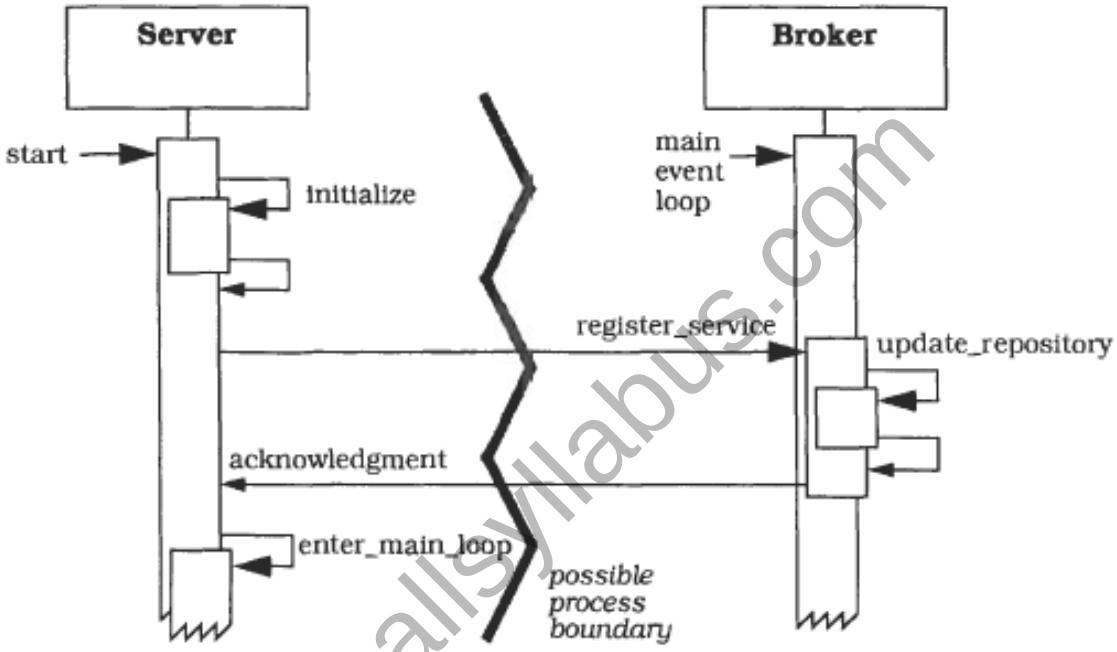
Relation between objects:



Dynamics:

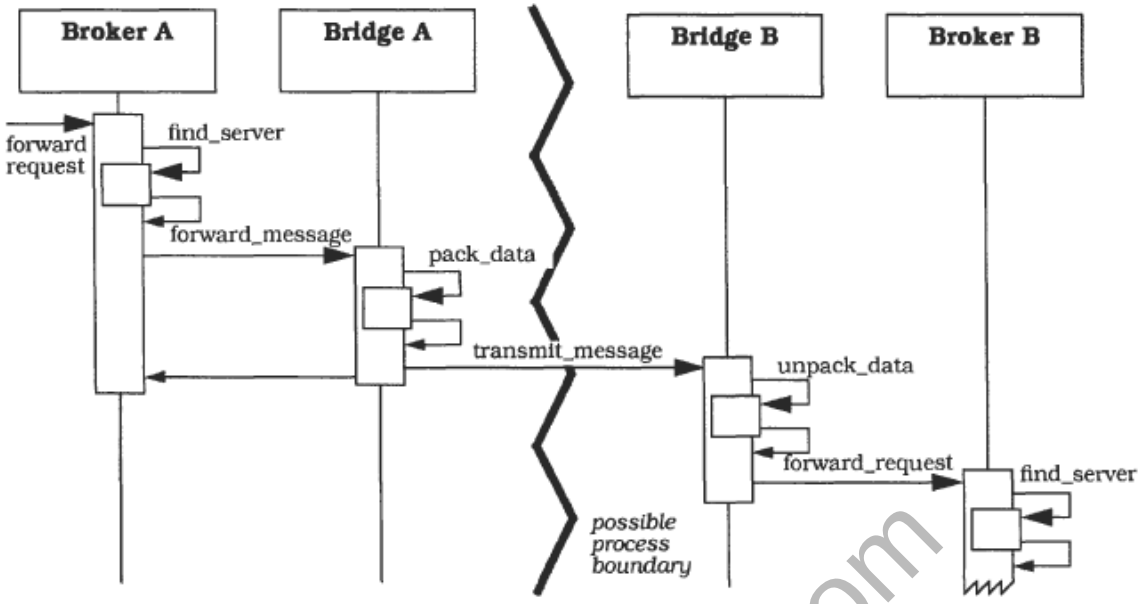
Scenario 1: Server registers itself with local broker component

- The broker is started in the initialization phase of the system. The broker enters its event loop and waits for incoming messages.
- The user starts a server application. Server executes initialization code and then registers with a broker
- The broker receives the incoming registration request from the server. It extracts all necessary information from the message and stores it into one or more repositories. These repositories are used to locate and activate servers. An acknowledgment is sent back.
- After receiving the acknowledgment from the broker, the server enters its main loop waiting for incoming client requests.



Scenario 2: Interaction between different brokers via bridge

- Broker A receives an incoming request. It locates the server responsible for executing the specified service by looking it up in the repositories. Since the corresponding server is available at another network node, the broker forwards the request to a remote broker.
- The message is passed from Broker A to Bridge A. This component is responsible for converting the message from the protocol defined by Broker A to a network-specific but common protocol understood by the two participating bridges. After message conversion, Bridge A transmits the message to Bridge B.
- Bridge B maps the incoming request from the network-specific format to a Broker B-specific format.
- Broker B performs all the actions necessary when a request arrives, as described in the first step of this scenario.



Implementation

Step1: *Define an object model, or use an existing model.*

- Each object model must specify entities such as object names, objects, requests, values, exceptions, supported types, type extensions, interfaces and operations.
- Only Semantic issues are considered in this step
- If the object model has to be extensible, prepare the system for future enhancements.
- The description of the underlying computational model is a key issue in designing an object model.
- We need to describe definitions of the state of server objects, definitions of methods, how methods are selected for execution and how server objects are generated and destroyed.

Step2: *Decide which kind of component-interoperability the system should* offer.

- We can design for interoperability either by specifying a binary standard or by introducing a high-level interface definition language (IDL).
- An IDL file contains a textual description of the interfaces a server offers to its clients.
- An IDL compiler uses an IDL file as input and generates programming-language code or binary code.
- One part of this generated code is required by the server for communicating with its local broker, another part is used by the client for communicating with its local broker.

Step3: *Specify the APIs the broker component provides for collaborating with clients and servers.*

- On the client side, functionality must be available for constructing requests, passing them to the broker and receiving responses.
- Decide whether clients should only be able to invoke server operations statically, allowing clients to bind the invocations at compile-time.
- If you want to allow dynamic invocations of servers as well, this has a direct impact on the size or number of APIs.
- If clients, servers and the broker are running as distinct processes, the API functions need to be based on an efficient mechanism for inter-process communication between clients, servers and the local broker.

Step4: *Use proxy objects to hide implementation details from clients and servers.*

- On the client side, a local proxy object represents the remote server object called by the client. On the server side, a proxy is used for playing the role of the client.
- Client-side proxies package procedure calls into messages and forward these messages to the local broker component.
- Server-side proxies receive requests from the local broker and call the methods in the interface implementation of the corresponding server.
- Proxies hide implementation details by using their own inter-process communication mechanism to communicate with the broker component.

Step5: *Design the broker component in parallel with steps 3 and 4.*

- During design and implementation, iterate systematically through the following steps:
  - Specify a detailed on-the-wire protocol for interacting with client-side proxies and server-side proxies.
  - A local broker must be available for every participating machine in the network.
  - When a client invokes a method of a server, the Broker system is responsible for returning all results and exceptions back to the original client.
  - If the proxies do not provide mechanisms for marshaling and unmarshaling parameters and results, you must include that functionality in the broker component.
- If your system supports asynchronous communication between clients and servers, you need to provide message buffers within the broker or within the proxies for the temporary storage of messages.
- Include a directory service for associating local server identifiers with the physical location of the corresponding servers in the broker.
- When your architecture requires system-unique identifiers to be generated dynamically during server registration, the broker must offer a name service for instantiating such names.
- If your system supports dynamic method invocation, the broker needs some means for maintaining type information about existing servers.
- Plan the broker's actions when the communication with clients, other brokers or servers fails.

Step6: *Develop IDL compilers*

- Whenever you implement interoperability by providing an interface definition language, you need to build an *IDL* compiler for every programming language you support**.**
- An IDL compiler translates the server interface definitions to programming language code.

Consequences - Advantages

- The Broker architectural pattern has some important **benefits:**
  - *Location Transparency.* As the broker is responsible for locating a server by using a unique identifier, clients do not need to know where servers are located.
  - *Changeability and extensibility of components.* If servers change but their interfaces remain the same, it has no functional impact on clients.

- *Portability of a Broker system.* The Broker system hides operating system and network system details from clients and servers by using indirection layers such as APIs, proxies and bridges.
- *Reusability*. When building new client applications, you can often base the functionality of your application on existing services

Liabilities:

- *Restricted efficiency.* Applications using a Broker implementation are usually slower than applications whose component distribution is static and known.
- *Lower fault tolerance.* Compared with a non-distributed software system, a Broker system may offer lower fault tolerance.

What is a Pattern

• A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and represents a well proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate

– Buschmann, F. et al, Pattern Oriented Software Architecture – Volume1, Wiley, 1996

Properties of Patterns

• Addresses a recurring design problem that arises in specific design situations and presents a solution to it

• Document existing, well-proven design experience

• Identify and Specify abtractions at the high(est) level

• Provide a common vocabulary and understanding design problems

ARCHITECTURAL PATTERN

THREE PART SCHEMA

1.CONTEXT

2.PROBLEM

3.SOLUTION

ADDITIONAL PARTS

VARIANTS

BENEFITS

LIABILITIES

Design situation giving rise to a design problem

CONTEXT

• Describe situations in which the problem occurs

• Specifying all situations is practically not possible

– List all known situations; provides guidance

• Example

– Developing Messaging solution for a mobile phone

– Developing software for a Man Machine Interface

PROBLEM

Set of forces repeatedly arising in the context

• Starts with a generic problem statement; captures the central theme

• Completed by *forces;* aspect of the problem that should be considered when solving it

– Requirements

Context

– Constraints

– Desirable properties

• Forces complement or contradict

SOLUTION

Configuration to balance forces

– Structure with components and relationships

– Run-time behaviour

• Structure: A spatial behaviour – addresses static part of the solution

• Run-time: Behaviour while running – addresses the dynamic part

• Example

– Building blocks for the application

– Specific inputs events and their processing

Category Description

Architectural An architectural pattern expresses a *fundamental structural organization* schema of software systems. It provides a set of predefined subsystems, specifies their relationships, and includes rules and guidelines for organising the relationships between them.

Design A design pattern provides a scheme for *refining the subsystems or components* of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

Idiom An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationship between them using features of the given language.

Category Description

Mud to Structure Includes patterns that support suitable decomposition of an overall system task into cooperating subtasks

Distributed Systems Includes patterns that provide infrastructures for systems that have components located in different processes or in several subsystems and components

Interactive Systems Includes patterns that help to structure human-computer interaction

Adaptable Systems Includes patterns that provide infrastructures for the extension and adaptation of application in response to evolving and changing functional requirements

Structural Decomposition Includes patterns that support a suitable decomposition of subsystems and complex components into cooperating parts

ARCHITECTURAL PATTERNS

INTERACTIVE SYSTEMS

1.MODEL VIEW CONTROLLER

2.PRESENTATION ABSTRACTION CONTROL

ADAPTIVE SYSTEMS

1.MICROKERNAL

2.REFLECTION

ARCHITECTURAL PATTERNS

MUD TO STRUCTURE

1.PIPE AND FILTER

2.LAYERS

3.BLACK BOARD

DISTRIBUTED SYSTEMS

1.BROKER

1. **INTERACTIVE SYSTEMS**

   PATTERNS THAT HELPS TO STRUCTURE HUMAN COMPUTER INTERACTION

   2.**ADAPTABLE SYSTEMS**

   PROVIDES INFRASTRUCTURE FOR EXTENSION AND ADOPTION OF APPLICATION TO CHANGING FUNCTIONAL REQUIREMENTS

PATTERN DISCUSSION CLASSIFICATION

1.CONTEXT

2.PROBLEM

3.SOLUTION

4.BENEFITS

5.LIABILITIES

6.VARIANTS

MVC – MODEL VIEW CONTROLLER

DIVIDES INTERACTIVE APPLICATION INTO 3 PARTS

1. MODEL – CONTAINS CORE FUNCTIONALITY AND DATA

2. VIEW – DISPLAY INFORMATION TO USER.

3. CONTROLLER – HANDLES USER INPUT.

VIEW AND CONTROLLER TOGETHER GIVES USER INTERFACE.

EXAMPLE : GRADES IN THE FORM OF DIFFERENT TYPES OF CHARTS AND TABLES.

HOW MVC WORKS ?

1.EVENT IS PASSED TO CONTROLLER

2.CONTROLLER CHANGES VIEWS

3.VIEWS GET DATA FROM MODEL

4.MODEL UPDATES VIEW WHEN DATA CHANGES.

IMPLEMENTATION

FUNDAMENTAL STEPS :

1. SEPARATE HUMAN COMPUTER INTERACTION FROM CORE FUNCTIONALITY.

2.IMPLEMENT CHANGE PROPAGATION MECHANISM.

3.DESIGN AND IMPLEMENT VIEWS.

4.DESIGN AND IMPLEMENT CONTROLLERS

5.DESIGN AND IMPLEMENT VIEW CONTROLLER RELATIONSHIP.

6. IMPLEMENT SETUP OF MVC.

ADDITIONAL STEPS

HIGHER DEGREE OF FREEDOM AND FLEXIBILITY

DYNAMIC VIEW CREATION

PLUGGABLE CONTROLLERS

HIERARCHICAL VIEWS AND CONTROLLERS.

DECOUPLING FROM SYSTEM DEPENDENCIES.

VARIANTS :

LOOSE COUPLING OF VIEW AND CONTROLLER ENABLES DIFFERENT VIEWES OF THE SAME DOCUMENT.

BENEFITS:

MULTIPLE VIEWS OF THE SAME MODEL

PLUGGABLE VIEW AND CONTROLLER

LIABILITIES:

INCREASED COMPLEXITY

INEFFICIENCY OF DATA ACCESS IN VIEW.

INTERACTIVE SYSTEMS

1. MODEL VIEW CONTROLLER(MVC)

2. PRESENTATION ABSTRACTION CONTROL(PAC)

HIERARCHY OF COOPERATING AGENTS

EACH AGENT IS RESPONSIBLE FOR APPLICATIONS FUNCTIONALITY

CONSISTS OF THREE COMPONENTS , PRESENTATION ABSTRACTION AND CONTROL

THIS SEPARATES HUMAN COMPUTER INTERACTION WITH CORE FUNCTIONALITY

PAC IS USED WHERE MANY SUB SYSTEMS ARE INVOLVED IN THE APPLICATION
ORGANISES THE COMMUNICATION BETWEEN CORE FUNCTIONALITY AND USER INTERFACE
WHICH IS NOT ADDRESSED IN MVC.

CONTEXT : DEVELOPMENT OF AN INTERACTIVE APPLICATION BY USING AGENTS.
PROBLEM : INTERACTIVE SYSTEMS SHOULD BE VIEWED AS A SET OF COOPERATING AGENTS , AGENTS SHOULD MAINTAIN THEIR STATE AND DATA ,
AGENTS SHOULD PROVIDE THEIR OWN UI, SOLUTION : STRUCTURE THE APPLICATION LIKE A TREE OF PAC AGENTS.EACH AGENT DEPENDS ON ITS TOP LEVEL AGENTS.

AGENTS
1. PRESENTATION COMPONENT PROVIDES VISIBLE BEHAVIOR
2.ABSTRACTION COMPONENT MAINTAINS DATA MODEL
3.CONTROL COMPONENTS CONNECTS PRESENTATION AND ABSTRACTION COMPONENTS.
TOP LEVEL PAC AGENT PROVIDES FUNCTIONAL CORE OF THE SYSTEM ,
BOTTEM LEVEL PAC AGENT PROVIDES SEMANTICS OF THE SYSTEM,INTERMEDIATE LEVEL AGENTS MAINTAINS COMMUNICATION BETWEEN LOWER LEVEL AGENTS.

DYNAMICS
SCENERIOS USE SEQUENCE DIAGRAMS IN ITS SIMPLEST FORM.
IMPLEMENTATION
1. DEFINE A MODEL OF THE APPLICATION
2.DEFINE A STRATEGY FOR DESIGNING PAC HIERARCHY
3.SPECIFY TOP LEVEL PAC AGENT.
4.SPECIFY BOTTOM LEVEL PAC AGENTS FOR SYSTEM SERVICES.
5.SPECIFY INTERMEDIATE LEVEL PAC AGENTS.
6.SEPARATE CORE FUNTIONALITY WITH HUMAN INTERFACE.

7.INTRODUCE THE CONTROL COMPONENT.
8.LINK THE HIERARCHY TOGETHER.
VARIANTS : PAC AGENTS AS ACTIVE OBJECTS.
PAC AGENTS AS ACTIVE PROCESSES.
BENEFITS:
SEPERATION OF CONCERNS , SUPPORT FOR CHANGE AND EXTENSION,SUPPORT FOR MULTI TASKING
LIABILITIES :INCREASED SYSTEM COMPLEXITY ,COMPLEX CONTROL COMPONENT , OVERHEAD IN COMMUNICATION ,SMALLER ATOMIC SEMANTIC CONCEPTS