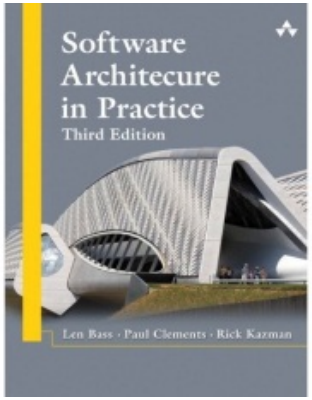
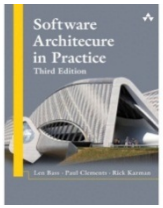
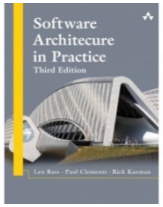


Chapter 13: Patterns and Tactics

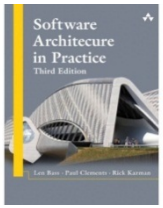

Software
Architecure
in Practice
Third Edition


昵图网 nipic.com


Software
Architecure
in Practice
Third Edition

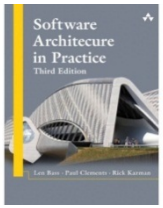

昵图网 www.nipic.com
BY: li.dian
NO:20100429165657092427


大展鴻圖
昵图网 www.nipic.com
By:义乌宏特 No.20120616180303376000



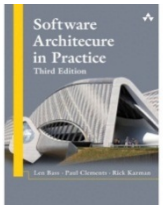
Applying Architectural Patterns

- RMI
- Web service (SOA)
- Peer-to-Peer
- MapReduce
- Layered
- MVC
- Publish-Subscribe



Chapter Outline

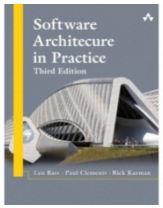
- What is a Pattern?
- Pattern Catalogue
  - Module patterns
  - Component and Connector Patterns
  - Allocation Patterns
- Relation Between Tactics and Patterns
- Using tactics together
- Summary



What is a Pattern?

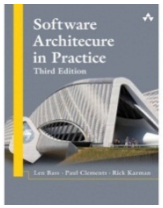
An architectural pattern establishes a relationship between:

- *A context*. A recurring, common situation in the world that gives rise to a problem.
- *A problem.* The problem, appropriately generalized, that arises in the given context.
- *A solution.* A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:
  - A set of element types (for example, data repositories, processes, and objects)
  - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
  - A topological layout of the components
  - A set of semantic constraints covering topology, element behavior, and interaction mechanisms

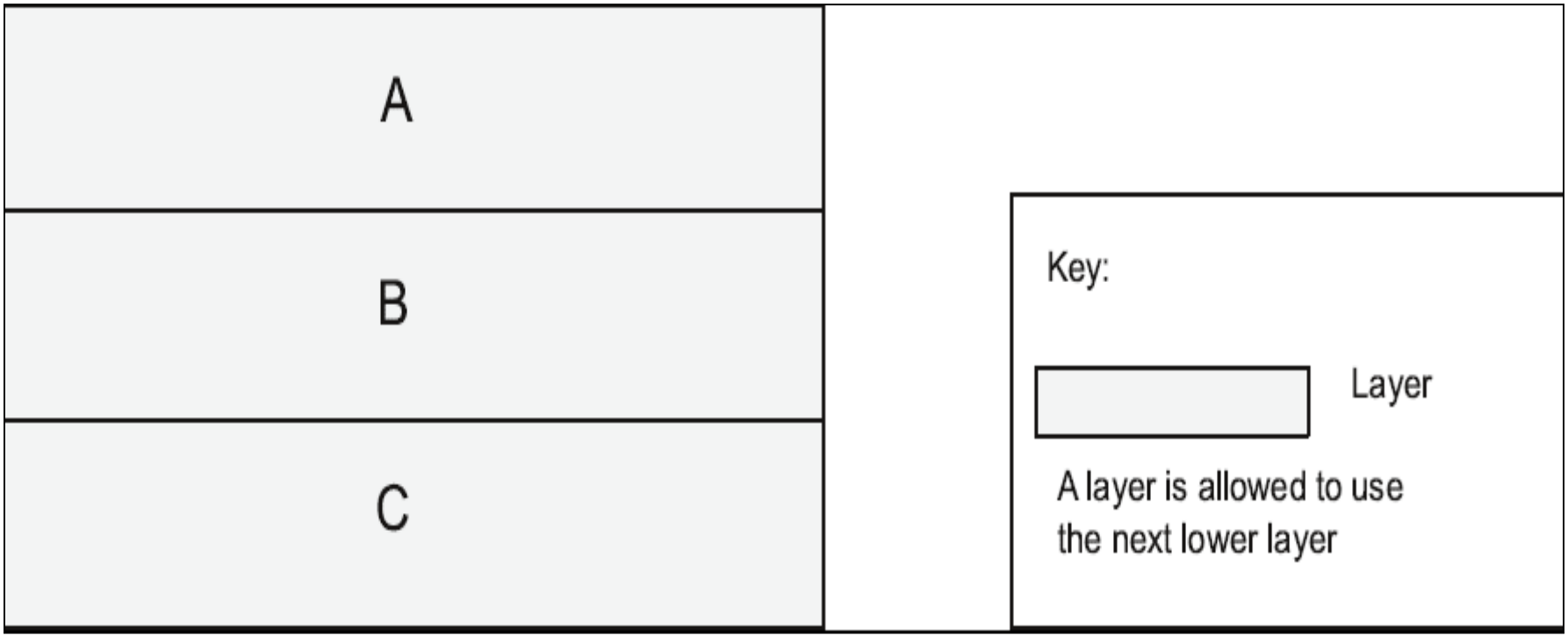


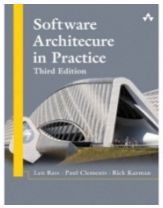
Layer Pattern

- **Context:** All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.
- **Problem:** The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting **portability, modifiability, and reuse**.
- **Solution:** To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.



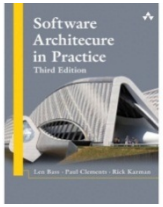
Layer Pattern Example





Layer Pattern Solution

- Overview: The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.
- Elements: *Layer*, a kind of module. The description of a layer should define what modules the layer contains.
- Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.
- Constraints:
  - Every piece of software is allocated to exactly one layer.
  - There are at least two layers (but usually there are three or more).
  - The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
- Weaknesses:
  - The addition of layers adds up-front cost and complexity to a system.
  - Layers contribute a performance penalty.

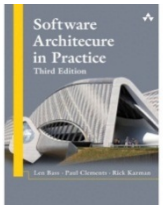


Exercises

- Use layer pattern to design the following cases

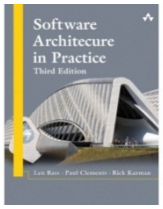
?

?

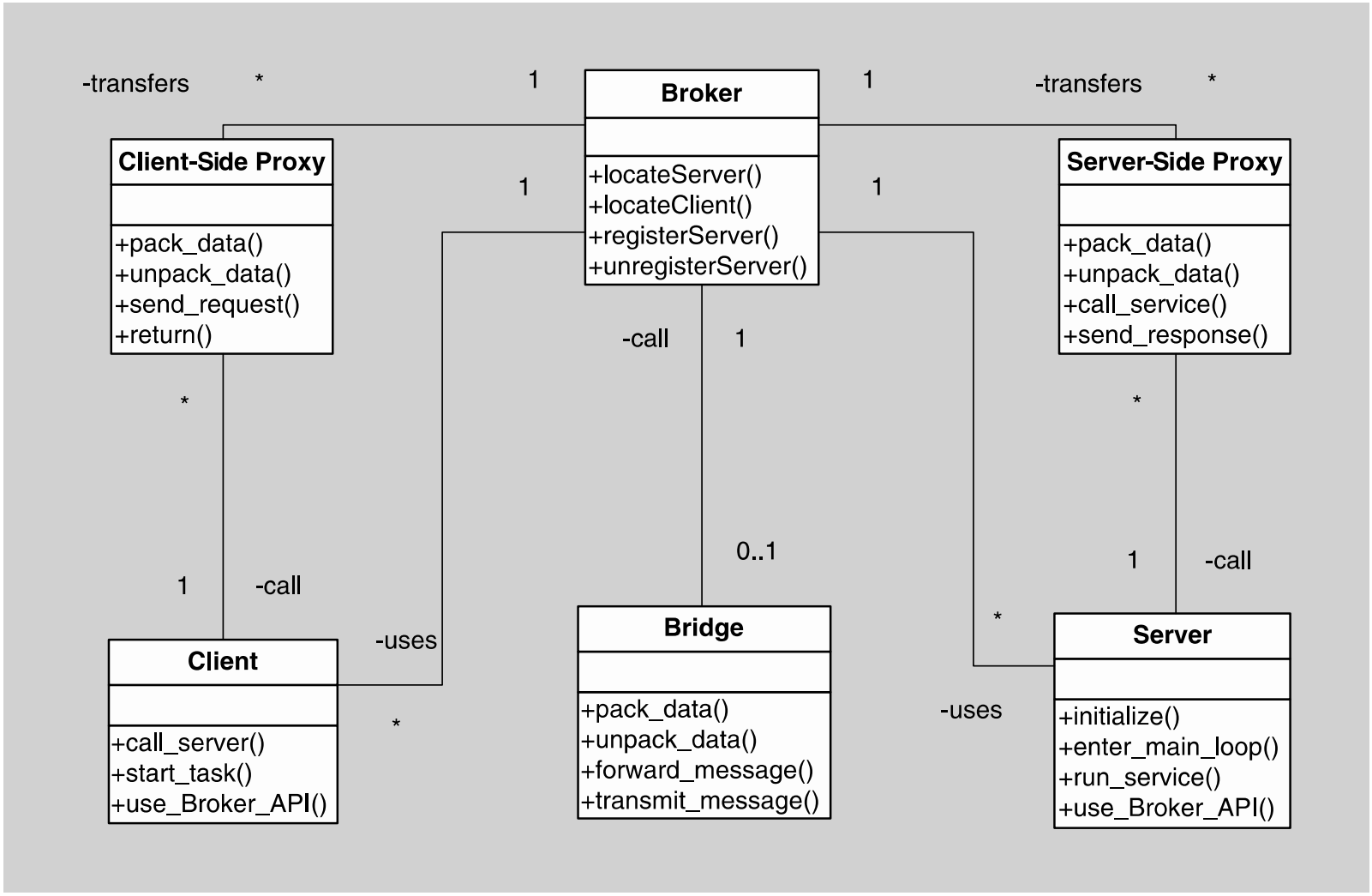


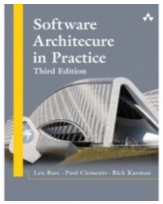
Broker Pattern

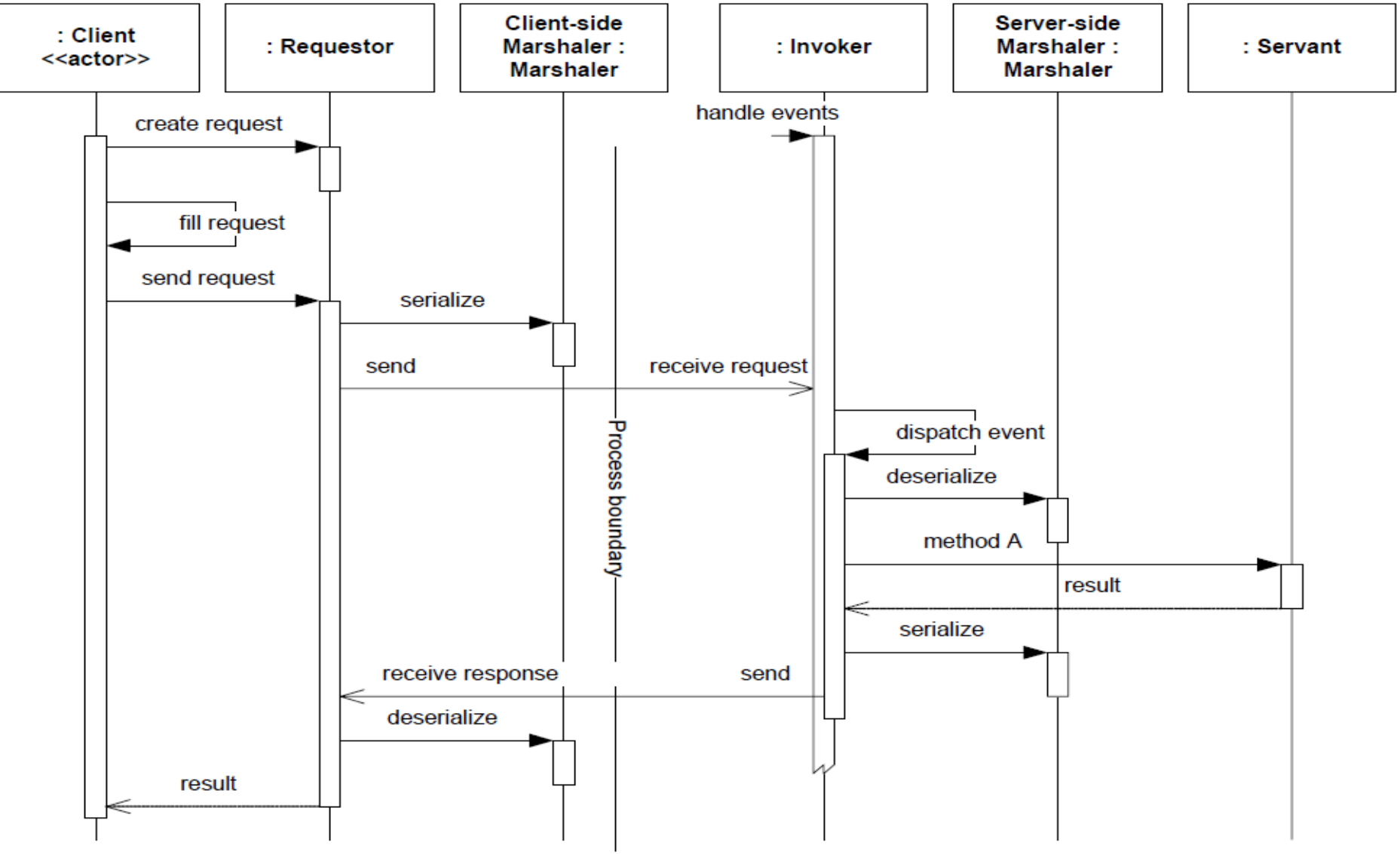
- **Context**: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.
- **Problem**: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?
- **Solution**: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

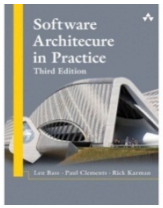


Broker Example



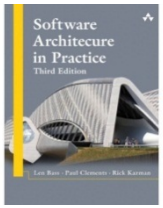

Software
Architecure
in Practice
Third Edition


: Client
<<actor>>
: Requestor
Client-side
Marshaler :
Marshaler
: Invoker
Server-side
Marshaler :
Marshaler
: Servant
handle events
create request
fill request
send request
serialize
send
receive request
Process boundary
dispatch event
deserialize
method A
result
serialize
receive response
send
deserialize
result



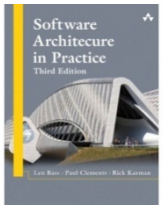
Broker Solution – 1

- Overview: The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.
- Elements:
  - *Client,* a requester of services
  - *Server,* a provider of services
  - *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client
  - *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
  - *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages



Broker Solution - 2

- Relations: The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
- Constraints: The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).
- Weaknesses:
  - Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
  - The broker can be a single point of failure.
  - A broker adds up-front complexity.
  - A broker may be a target for security attacks.
  - A broker may be difficult to test.

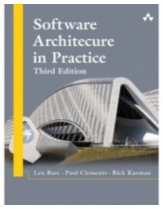


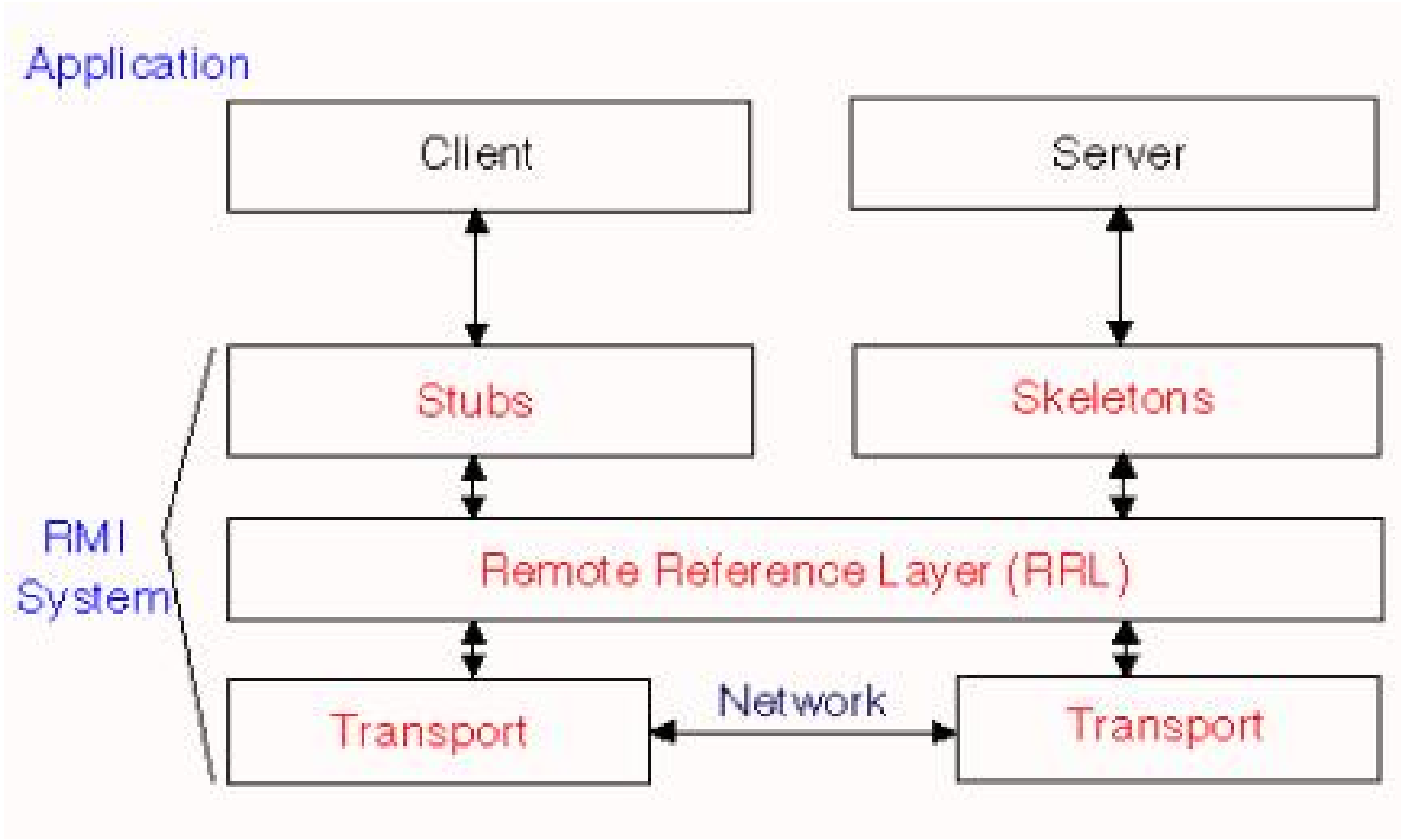
Exercises

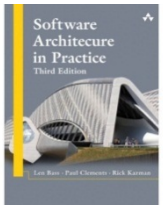
- Give an example of Broker pattern
- Use broker pattern to design the following cases

?

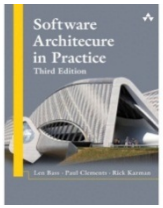
?


Software
Architecure
in Practice
Third Edition


Application
Client
Server
Stubs
Skeletons
RMI
System
Remote Reference Layer (RRL)
Transport
Network
Transport

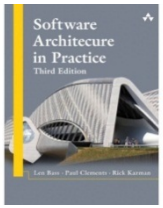


- RMI（Remote Method Invocation，远程方法调用）是用Java在JDK1.1中实现的，它大大增强了Java开发分布式应用的能力。可以被看作是RPC的Java版本。Java RMI 则支持存储于不同地址空间的对象之间彼此进行通信，实现远程对象之间的无缝远程调用。



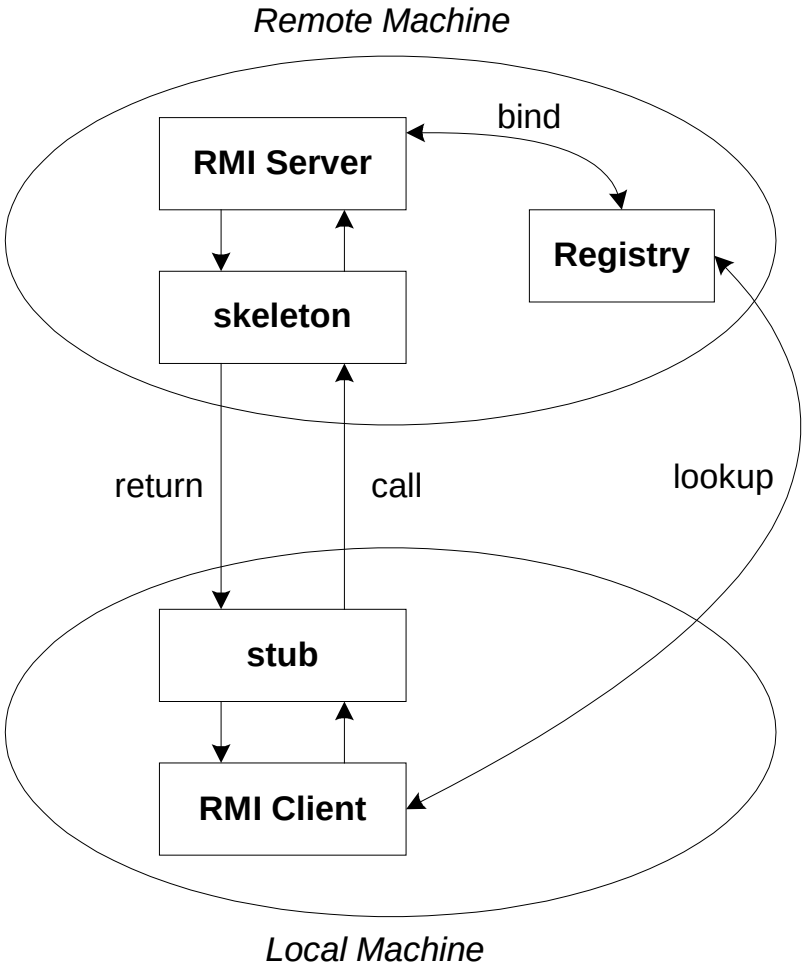
RMI系统运行机理

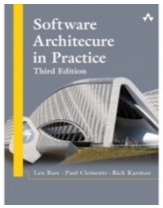
- RMI应用程序通常包括两个独立的程序：服务器程序和客户机程序。典型的服务器应用程序将创建多个远程对象，使这些远程对象能够被引用，然后等待客户机调用这些远程对象的方法。而典型的客户机程序则从服务器中得到一个或多个远程对象的引用，然后调用远程对象的方法。RMI为服务器和客户机进行通信和信息传递提供了一种机制。



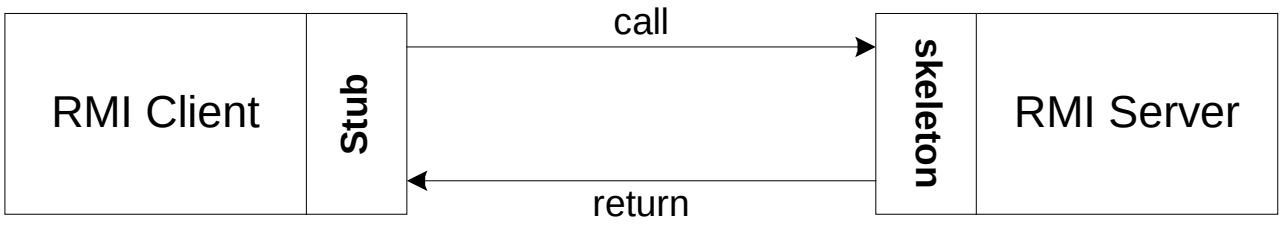
The General RMI Architecture

- The server must first bind its name to the registry
- The client lookup the server name in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.

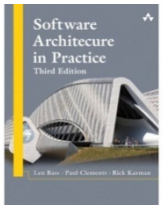




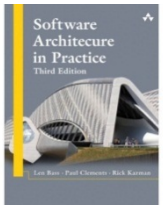
The Stub and Skeleton



- A client invokes a remote method, the call is first forwarded to stub.
- The stub is responsible for sending the remote call over to the server-side skeleton
- The stub opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton.
- A skeleton contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

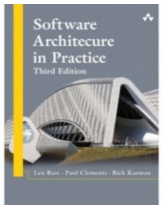


- 既然是分布式应用系统,就涉及多JRE的问题。在A JRE中的对象,它的内存地址是0xabcdef,该地址存放的值,和B JRE中的该地址放的值是不同的。那么如何确保对象的有效传递呢？

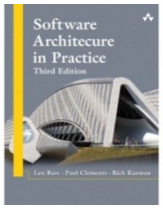


对象序列化机制

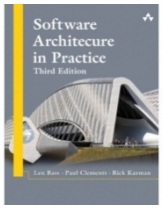
- 对象序列化机制（object serialization）是Java语言内建的一种对象持久化方式，可以很容易的在JVM中的活动对象和字节数组（流）之间进行转换。除了可以很简单的实现持久化之外，序列化机制的另外一个重要用途是在远程方法调用中，用来对开发人员屏蔽底层实现细节。



- 序列化和反序列化工作是通过ObjectOuputStream和ObjectInputStream来完成的。ObjectOutputStream的writeObject方法可以把一个Java对象写入到流中，ObjectInputStream的readObject方法可以从流中读取一个Java对象。

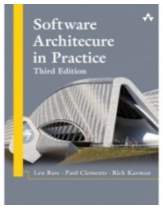



```
try {
    User user = new User("Alex", "Cheng");
    ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream("user.bin"));
    output.writeObject(user);
    output.close();
} catch (IOException e) {
    e.printStackTrace();}


try {
    ObjectInputStream input = new ObjectInputStream(new
FileInputStream("user.bin"));
    User user = (User) input.readObject();
    System.out.println(user);
} catch (Exception e) {
    e.printStackTrace();}
```



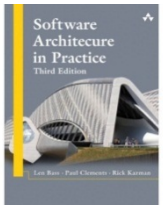
- 在写入和读取的时候，用的参数或返回值是单个对象，实际进行序列化和反序列化的时候是否仅仅这一个对象？

- 实际上操纵的是一个对象图，包括该对象所引用的其它对象，以及这些对象所引用的另外的对象。Java会自动帮你遍历对象图并逐个序列化。



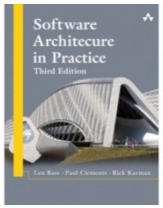
RMI（远程方法调用）的原理

- 在与远程对象的通信过程中，RMI使用标准机制：stub和skeleton。远程对象的stub担当远程对象的客户本地代理人角色。调用程序将调用本地stub的方法，而本地stub将负责执行对远程对象的方法调用。
- 在RMI中，远程对象的stub与该远程对象所实现的远程接口集相同。



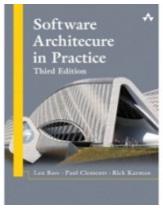
- 调用stub的方法时将执行下列操作：

(1) 初始化与包含远程对象的远程虚拟机的连接；(2) 对远程虚拟机的参数进行编组（写入并传输）；(3) 等待方法调用结果；(4) 解编（读取）返回值或返回的异常；(5) 将值返回给调用程序。为了向调用程序展示比较简单的调用机制，stub将参数的序列化和网络级通信等细节隐藏了起来。



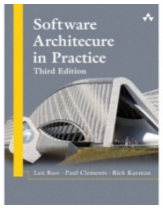
- 在远程虚拟机中，每个远程对象都可以有相应的skeleton（在JDK1.2及以上环境中无需使用skeleton）。Skeleton负责将调用分配给实际的远程对象实现。它在接收方法调用时执行下列操作：

  (1) 解编（读取）远程方法的参数；(2) 调用实际远程对象实现上的方法；(3) 将结果（返回值或异常）编组（写入并传输）给调用程序。stub和skeleton由rmic编译器生成。
- 从JDK5.0以后，这两个类就不需要rmic来产生了，而是有JVM自动处理，实际上他们还是存在的。



RMI具体实现

1.定义远程服务的接口
2.远程服务接口的具体实现
3.定义客户端



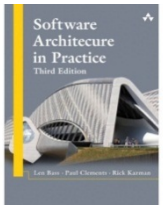
1. 将远程类的功能定义为Java接口

在Java中，远程对象是实现远程接口的类的实例。在远程接口中声明每个要远程调用的方法。

（1）远程接口必须声明为public。

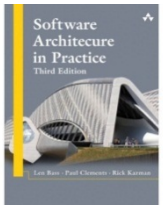
（2）远程对象扩展java.rmi.Remote接口。

（3）除了所有应用程序特定的例外之外，每个方法还必须抛出java.rmi.RemoteException例外。



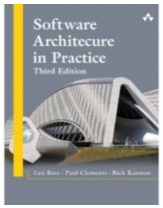
Step 1: Defining the Remote Interface

```
/* SampleServer.java */
import java.rmi.*;

public interface SampleServer extends Remote
{
  public int sum(int a,int b) throws RemoteException;
}
```



2.编写和实现服务器类。

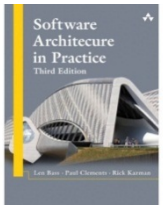
该类是实现1中定义的远程接口，并且必须具有构造方法。在该类中还要实现远程接口中所声明的各个远程方法。



Step 2: Develop the remote object and its interface

- The server is a simple unicast remote server.
- Create server by extending `java.rmi.server.UnicastRemoteObject`.

```
/* SampleServerImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class SampleServerImpl extends UnicastRemoteObject
                              implements SampleServer
{
  SampleServerImpl() throws RemoteException
  {
     super();
  }
```

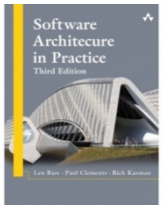


Step 2: Develop the remote object and its interface

- Implement the remote methods

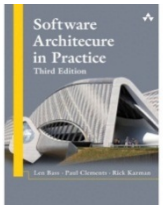
```
/* SampleServerImpl.java */
  public int sum(int a,int b) throws RemoteException
  {
     return a + b;
  }
}
```

- The server must bind its name to the registry, the client will look up the server name.
- Use `java.rmi.Naming` class to bind the server name to registry. In this example the name call “SAMPLE-SERVER”.



Step 2: Develop the remote object and its interface

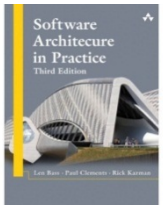
```
/* SampleServerImpl.java */
  public static void main(String args[])
  {
      try      {
        System.setSecurityManager(new RMISecurityManager());
        //set the security manager

        //create a local instance of the object
        SampleServerImpl Server = new SampleServerImpl();

        //put the local instance in the registry
        Naming.rebind("SAMPLE-SERVER" , Server);

        System.out.println("Server waiting.....");
      }
      catch (java.net.MalformedURLException me)       {
        System.out.println("Malformed URL: " +
   me.toString());   }
      catch (RemoteException re)  {
         System.out.println("Remote exception: " +
   re.toString());  }  }
```



3.编写使用远程服务的客户机程序。

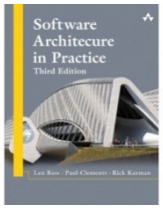
在该类中使用java.rmi.Naming中的lookup()方法获得对远程对象的引用，依据需要调用该引用的远程方法，其调用方式和对本地对象方法的调用相同。

- The server name is specified as URL in the from (rmi://host:port/name)
- Default RMI port is 1099.
- The name specified in the URL must exactly match the name that the server has bound to the registry. In this example, the name is “SAMPLE-SERVER”



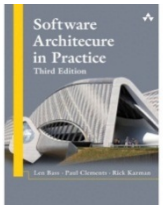
Step 3: Develop the client program

```
import java.rmi.*;
import java.rmi.server.*;
public class SampleClient
{
   public static void main(String[]  args)
   {
      // set the security manager for the client
      System.setSecurityManager(new RMISecurityManager());
      //get the remote object from the registry
      try
        {
          System.out.println("Security Manager loaded");
          String url = "//localhost/SAMPLE-SERVER";
          SampleServer remoteObject = (SampleServer)Naming.lookup(url);
          System.out.println("Got remote object");
          System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );
        }
        catch (RemoteException exc) {
          System.out.println("Error in lookup: " + exc.toString()); }
        catch (java.net.MalformedURLException exc) {
          System.out.println("Malformed URL: " + exc.toString());   }
        catch (java.rmi.NotBoundException exc)  {
          System.out.println("NotBound: " + exc.toString());
```



实现了服务器和客户机的程序后，就可以编译和运行该RMI系统。其步骤有：

1. 使用javac编译远程接口类，远程接口实现类和客户机程序。
2. 使用rmic编译器生成实现类的stub和skeleton。
3. 启动RMI注册服务程序rmiregistry。
4. 启动服务器端程序。
5. 启动客户机程序。



练习

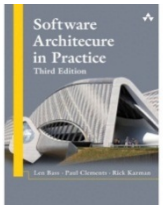
一.填空题

1.java.rmi.registry.Registry的默认侦听端口是___ 。

2.RMI有三个层次，分别为：_____、_____、_____。

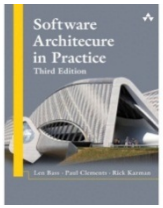
3.stub/skeleton层客户机和服务器之间的信息逻辑流。___是存储在客户机上的代理，它代表远程对象。客户机通过该代理与远程对象进行通信。而___是服务器上的代理。

4.编译stub和skeleton后，我们通过____来启动注册库。



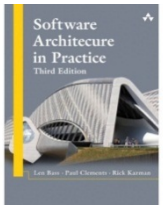
二.问答题

- 试简述创建RMI应用程序所涉及的步骤？
- 试述RMI应用中，stub与skeleton之间信息流的通信过程。

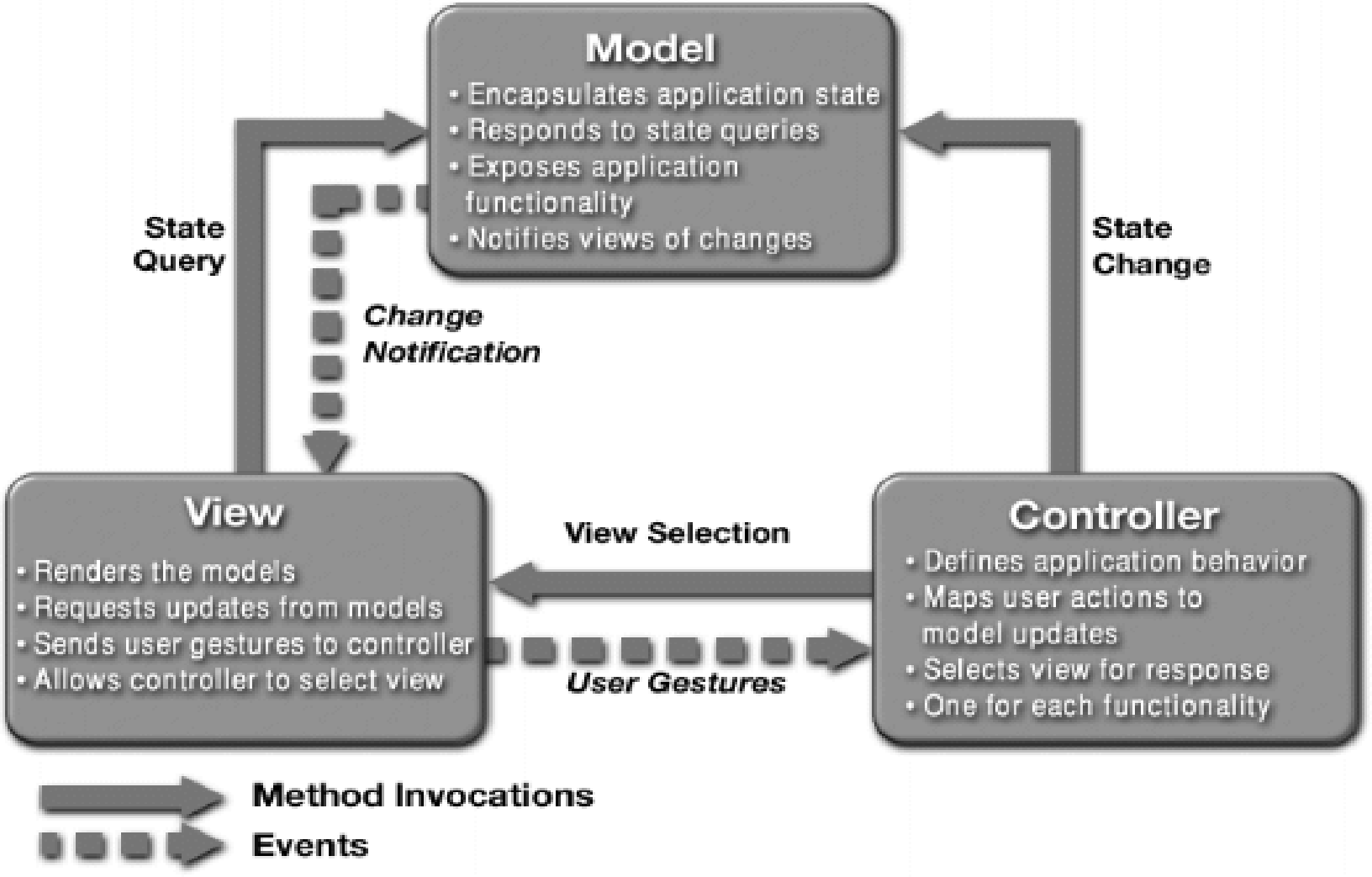


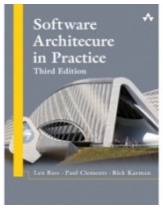
Model-View-Controller Pattern

- **Context**: User interface software is typically the most frequently modified portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.
- **Problem**: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?
- **Solution**: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:
  - A model, which contains the application's data
  - A view, which displays some portion of the underlying data and interacts with the user
  - A controller, which mediates between the model and the view and manages the notifications of state changes


Software
Architecure
in Practice
Third Edition

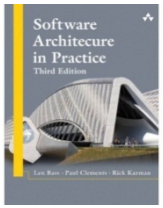
MVC Example


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events



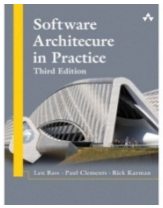
MVC Solution - 1

- Overview: The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
- Elements:
  - The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic.
  - The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.
  - The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.

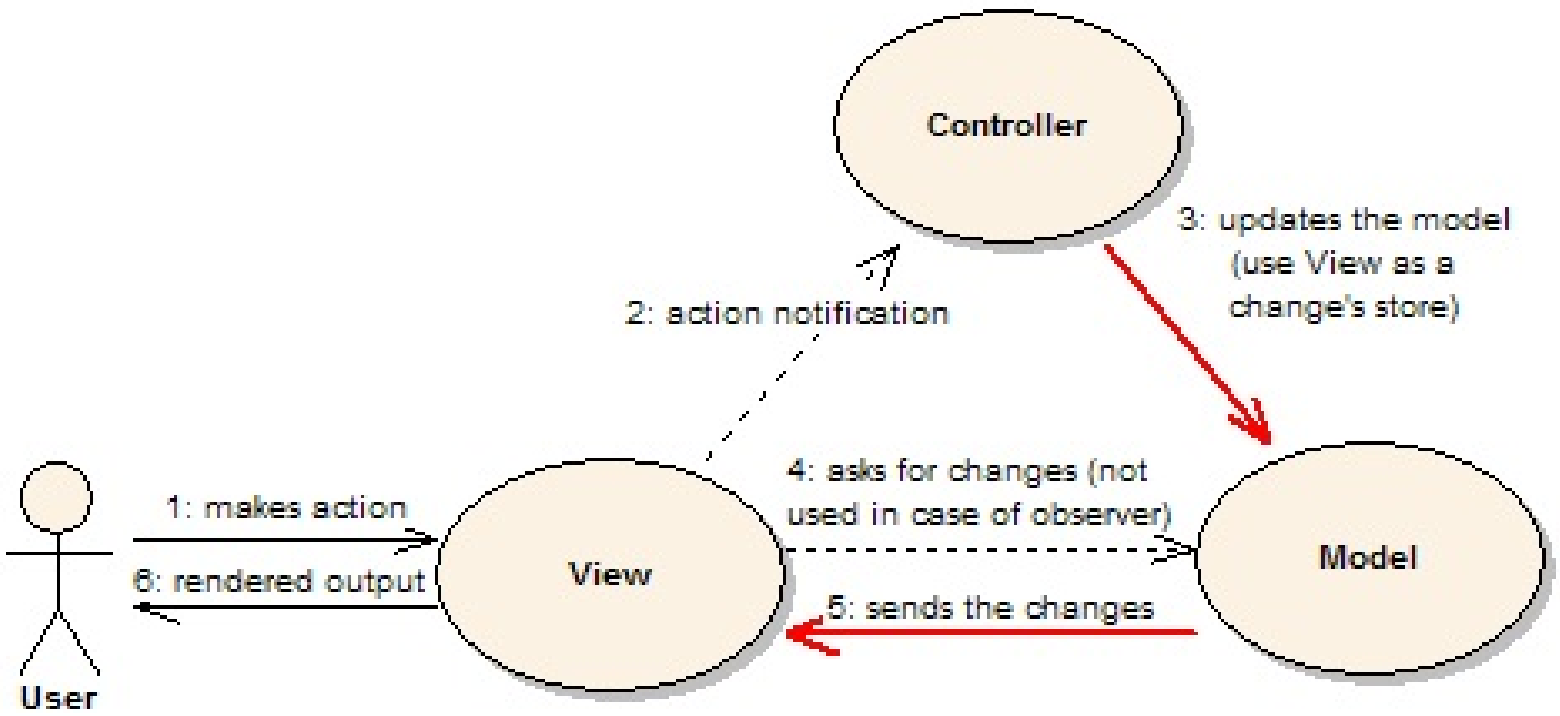


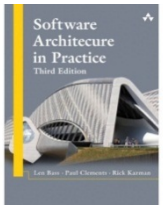
MVC Solution - 2

- Relations: The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
- Constraints:
  - There must be at least one instance each of model, view, and controller.
  - The model component should not interact directly with the controller.
- Weaknesses:
  - The complexity may not be worth it for simple user interfaces.
  - The model, view, and controller abstractions may not be good fits for some user interface toolkits.

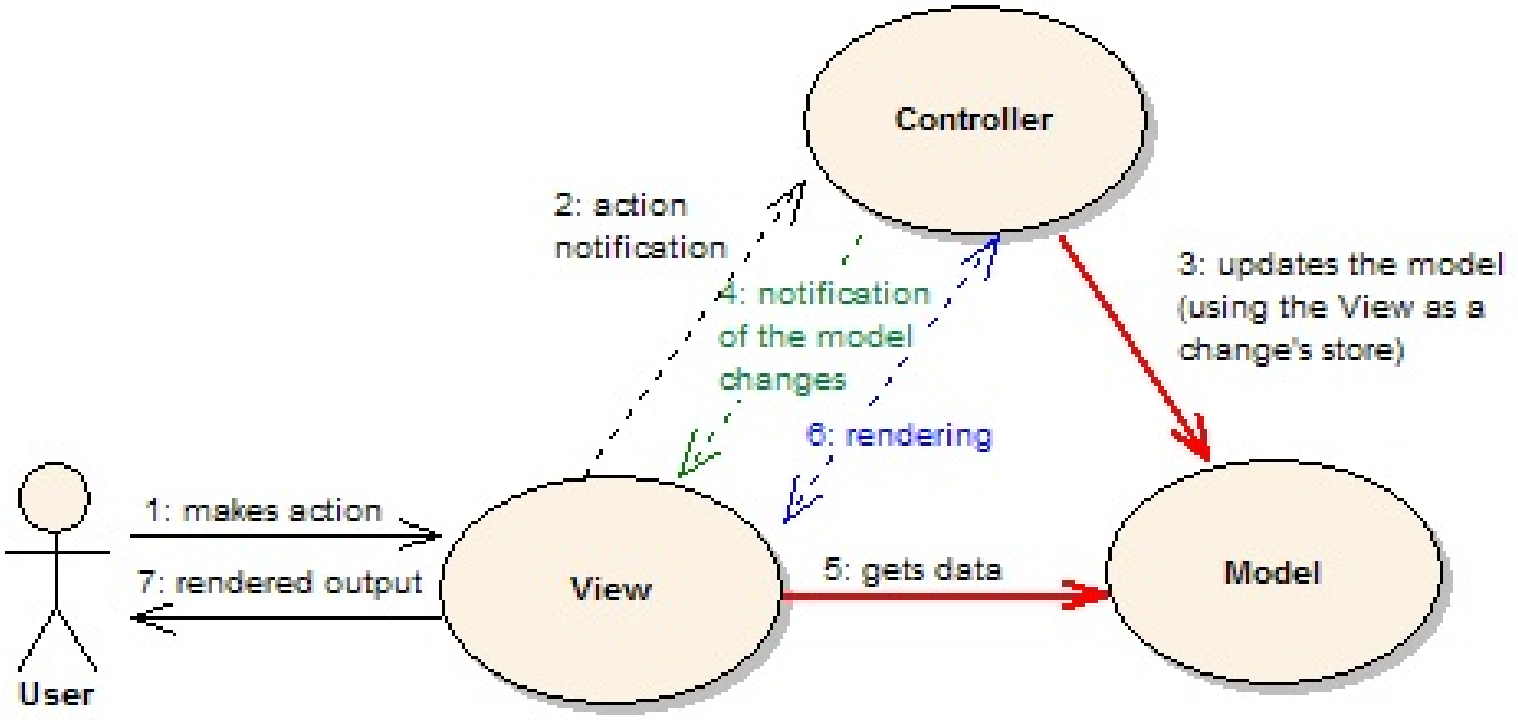

Software
Architecure
in Practice
Third Edition

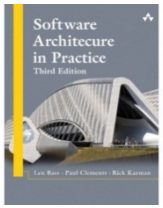
主动式MVC


Controller
3: updates the model
(use View as a
change's store)
2: action notification
4: asks for changes (not
used in case of observer)
1: makes action
6: rendered output
View
Model
5: sends the changes
User


Software
Architecure
in Practice
Third Edition

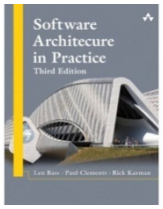
Passive MVC


Controller
2: action
notification
4: notification
of the model
changes
3: updates the model
(using the View as a
change's store)
6: rendering
1: makes action
7: rendered output
View
5: gets data
Model
User

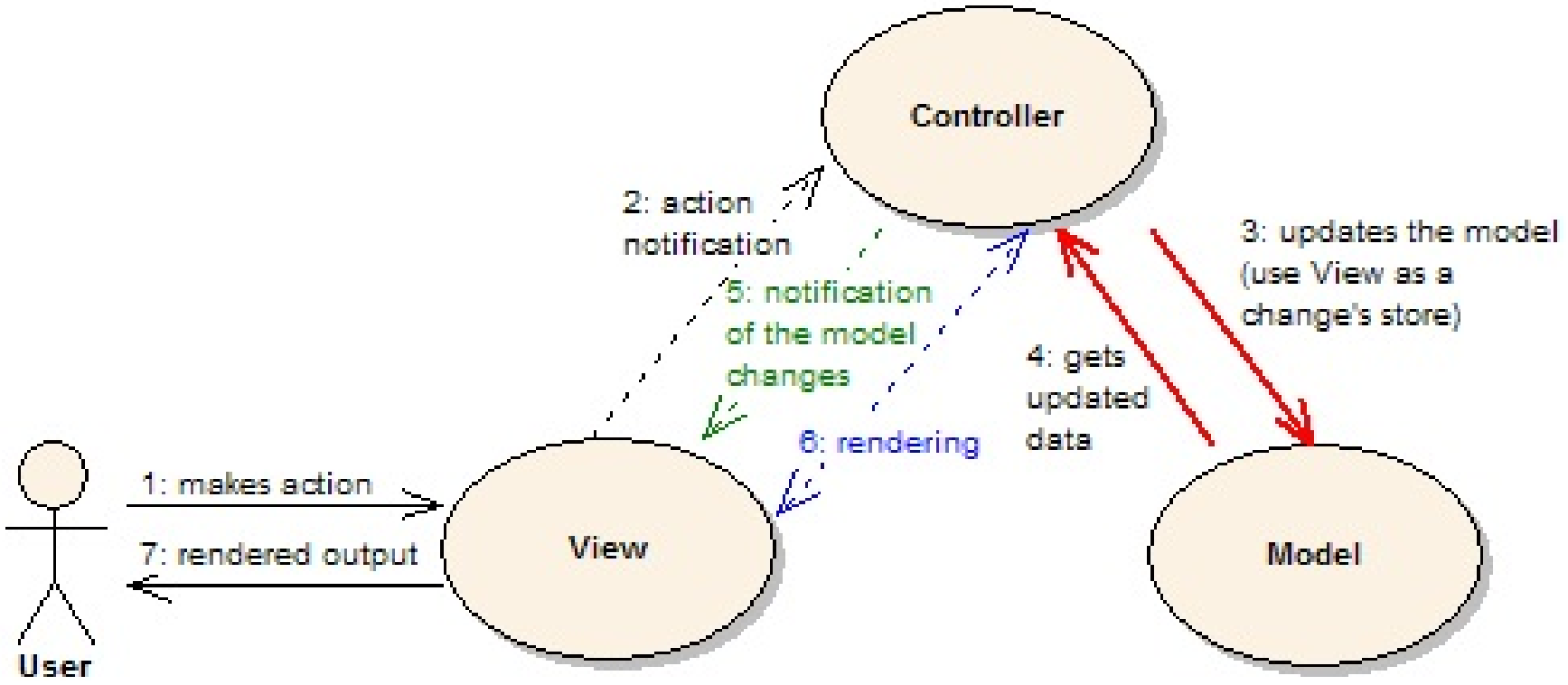


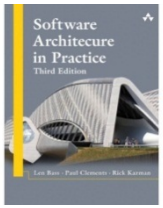
Shortcomings of MVC

- 在MVC里，View是可以直接访问Model的！从而，View里会包含Model信息，不可避免的还要包括一些业务逻辑。
  在MVC模型里，更关注的是Model的不变，即同时有多个对Model的不同显示View。
  所以，在MVC模型里，Model不依赖于View，但是View是依赖于Model的。不仅如此，因为有一些业务逻辑在View里实现了，导致要更改View也是比较困难的，至少那些业务逻辑是无法重用的。


Software
Architecure
in Practice
Third Edition

MVP


Controller
2: action
notification
5: notification
of the model
changes
3: updates the model
(use View as a
change's store)
4: gets
updated
data
6: rendering
1: makes action
7: rendered output
View
Model
User



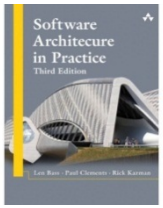
Exercises

- Give an example of MVC pattern
- Use MVC pattern to design the following cases

- See the examples

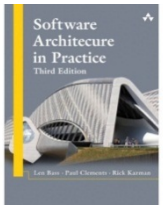
?

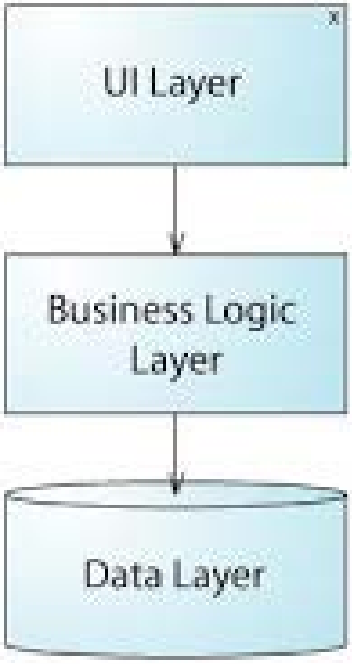

?

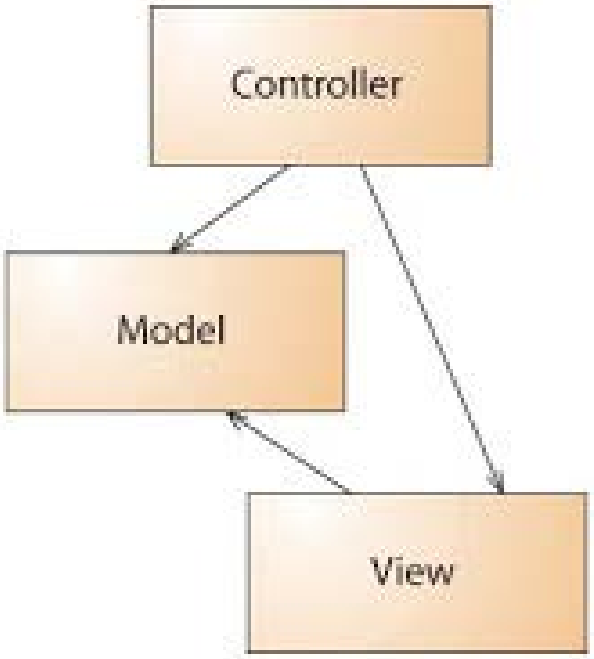


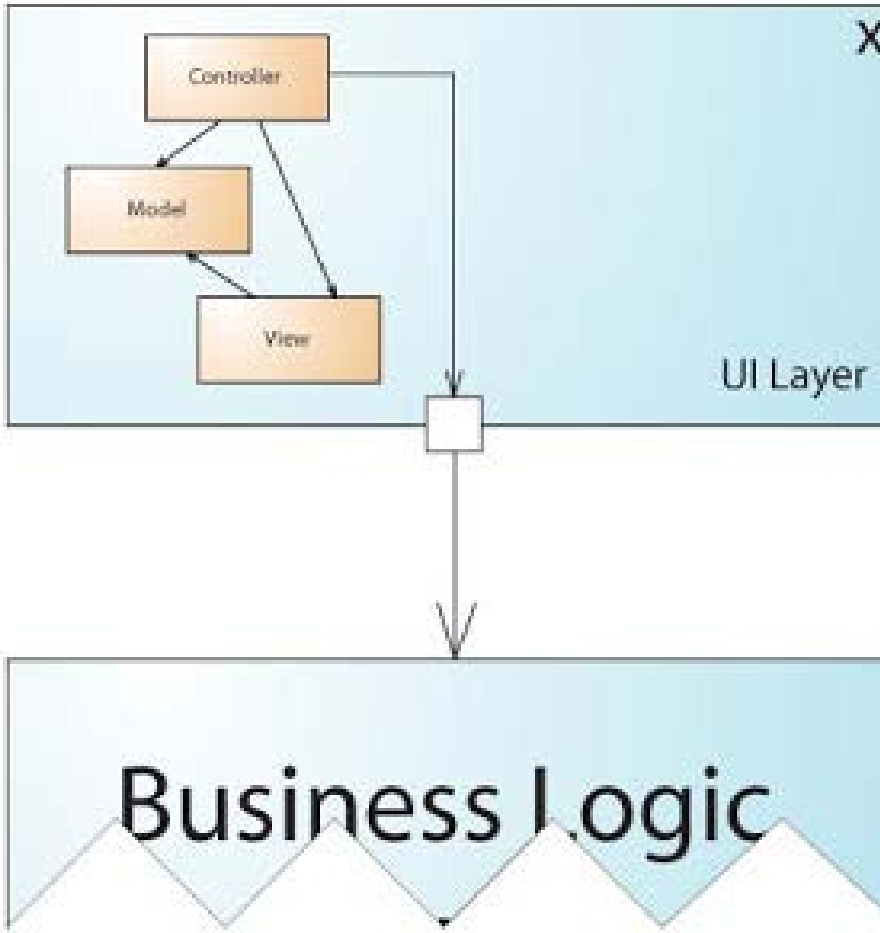
Questions

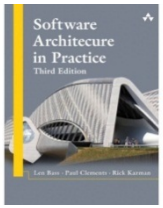
- What are the differences between Layered architecture and MVC?
- How to combine MVC and Layered architecture?
- **All UI modification (such as changing the text forecolor when a threshold is met) needs to be done in the view level as the model does not have any direct contact with controller. Is this correct?**
- Yes. If view is directly retrieving the new data from the Model.
- No. If view is seeking controllers/presenters' help to query the new data. In this case, the controller can format the data and send it to the view.


Software
Architecure
in Practice
Third Edition


UI Layer
Business Logic Layer
Data Layer

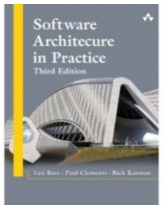

Controller
Model
View


Controller
Model
View
UI Layer
Business Logic

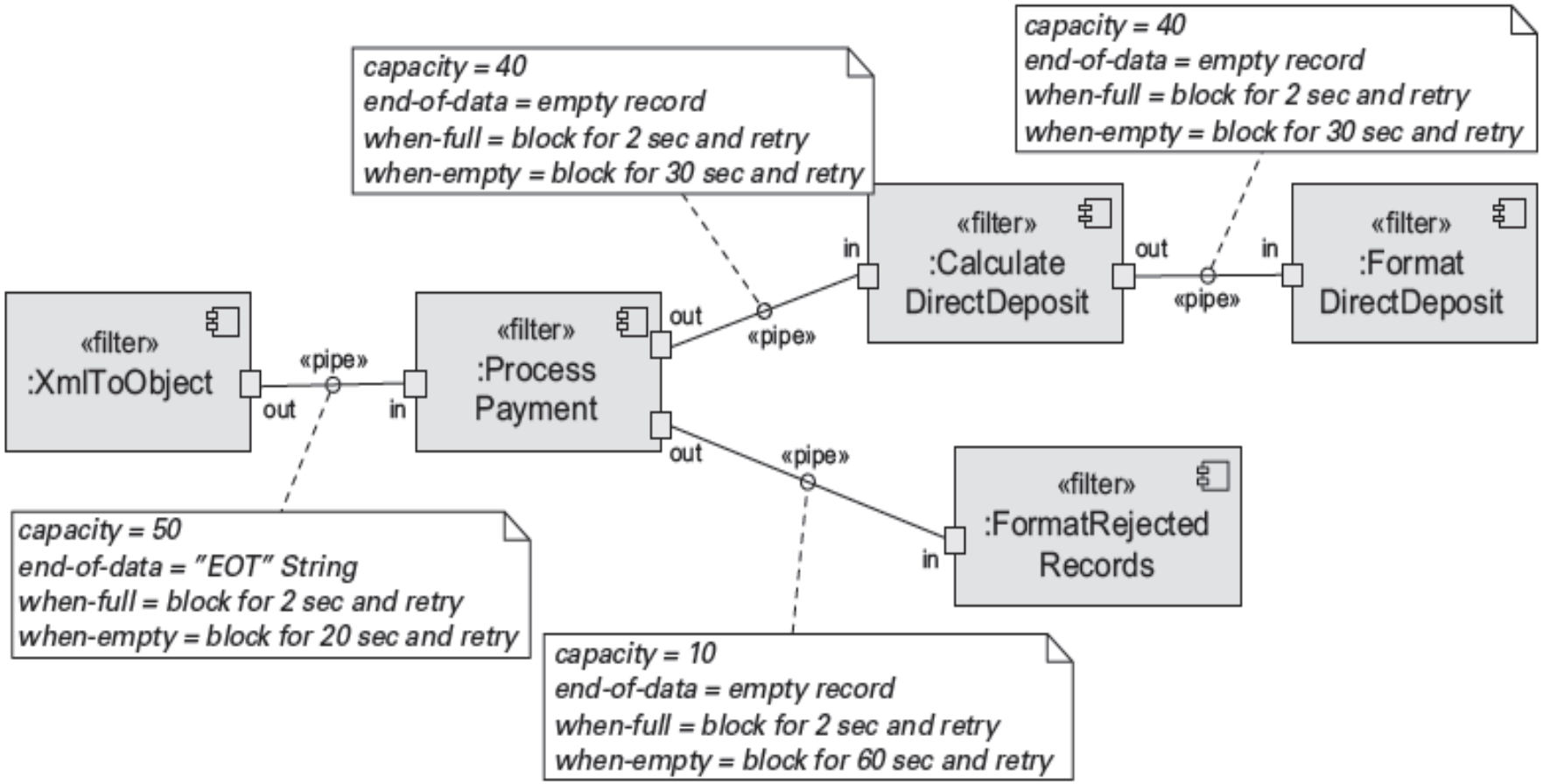


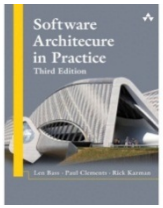
Pipe and Filter Pattern

- **Context:** Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.
- **Problem:** Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.
- **Solution:** The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.



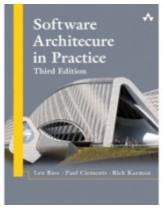
Pipe and Filter Example



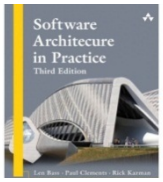


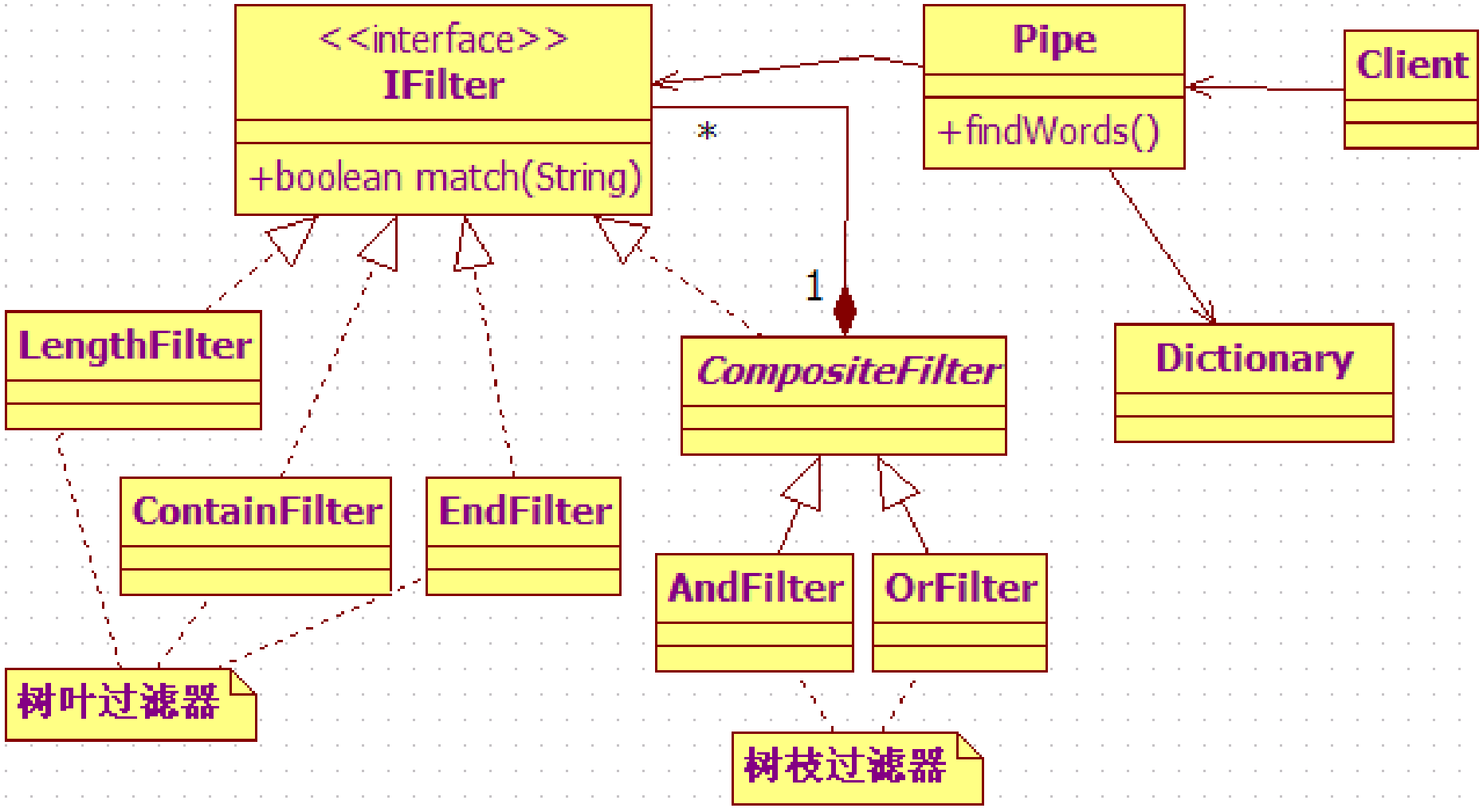
Pipe and Filter Solution

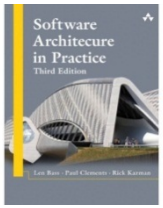
- Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
- Elements:
  - *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
  - *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.
- Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.
- Constraints:
  - Pipes connect filter output ports to filter input ports.
  - Connected filters must agree on the type of data being passed along the connecting pipe.



- 给定多个条件（即过滤器），遍历一本字典中的所有单词，将**同时符合所有条件**的所有单词查询（过滤）出来。现假定需要过滤出“**单词中同时包含a、b、cd字串，并且以"end”结尾，最后是单词的长度大于7**”。


Software
Architecure
in Practice
Third Edition


<<interface>>
IFilter
+boolean match(String)
Pipe
+findWords()
Client
*
1
LengthFilter
CompositeFilter
Dictionary
ContainFilter
EndFilter
AndFilter
OrFilter
树叶过滤器
树枝过滤器

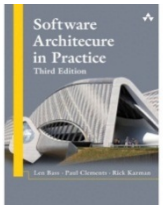


Exercises

- Give an example of Pipe&Filter pattern
- Use Pipe&Filter pattern to design the following cases

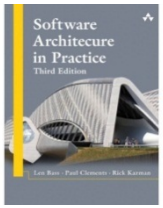
?

?

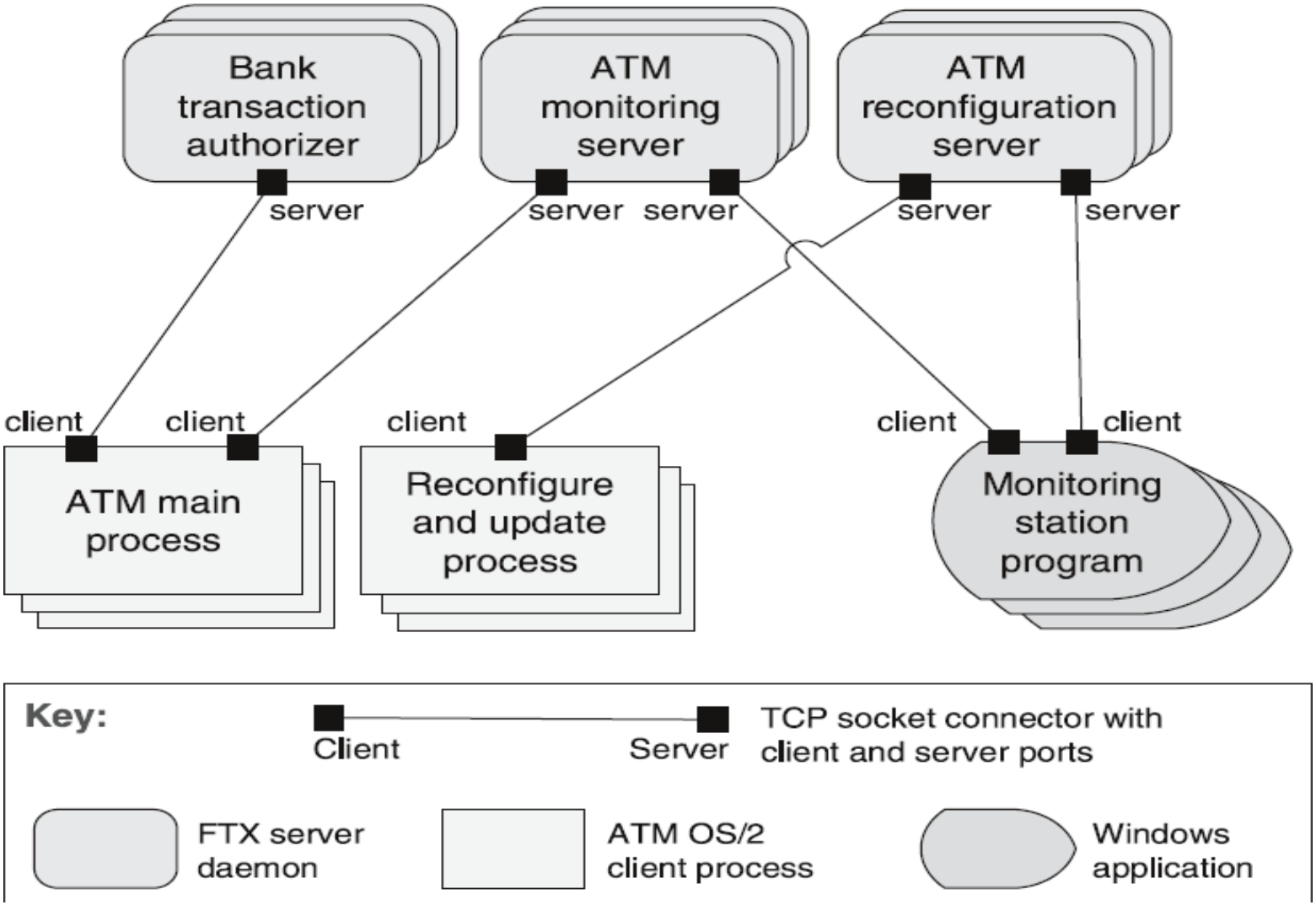


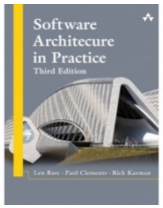
Client-Server Pattern

- **Context:** There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.
- **Problem:** By managing a set of shared resources and services, we can promote **modifiability** and **reuse**, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve **scalability** and **availability** by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.
- **Solution:** Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.



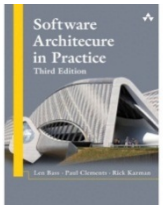
Client-Server Example





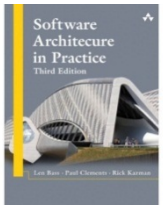
Client-Server Solution - 1

- Overview: Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
- Elements:
  - *Client,* a component that invokes services of a server component. Clients have ports that describe the services they require.
  - *Server:* a component that provides services to clients. Servers have ports that describe the services they provide.
- *Request/reply connector:* a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.



Client-Server Solution- 2

- Relations: The *attachment* relation associates clients with servers.
- Constraints:
  - Clients are connected to servers through request/reply connectors.
  - Server components can be clients to other servers.
- Weaknesses:
  - Server can be a performance bottleneck.
  - Server can be a single point of failure.
  - Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

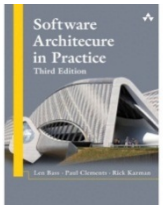


Exercises

- Give an example of Client-Server pattern
- Use Client-Server pattern to design the following cases

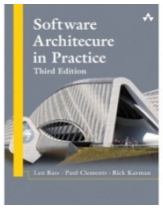
?

?

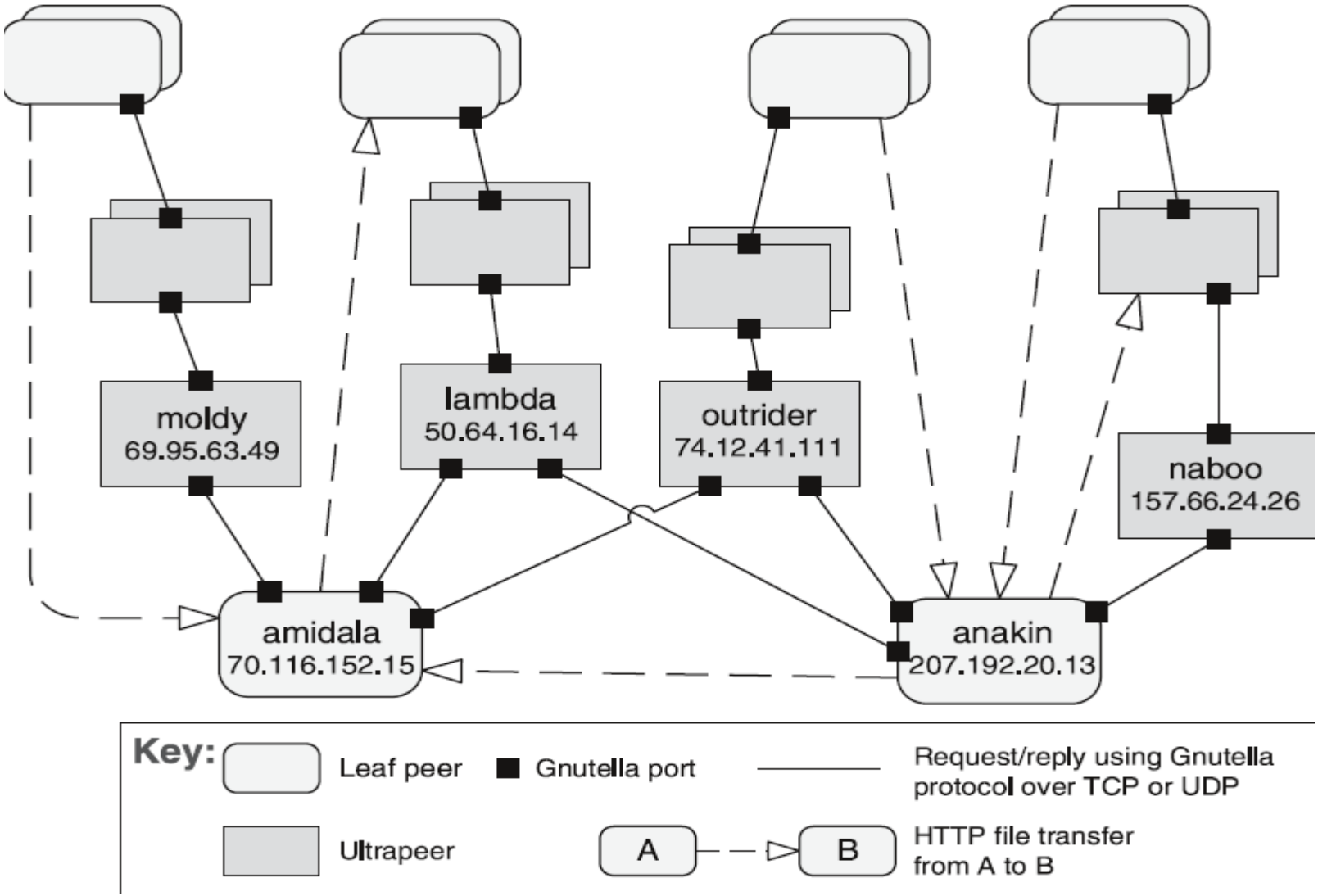


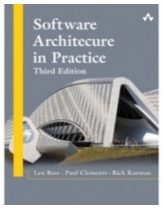
Peer-to-Peer Pattern

- **Context:** Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.
- **Problem:** How can a set of "equal" distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?
- **Solution:** In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are "equal" and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern.



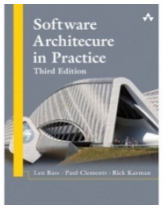
Peer-to-Peer Example





Peer-to-Peer Solution - 1

- Overview: Computation is achieved by cooperating peers that request service from and provide services to one another across a network.
- Elements:
  - *Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability.
  - *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
- Relations: The relation associates peers with their connectors. Attachments may change at runtime.



Peer-to-Peer Solution - 2

- Constraints: Restrictions may be placed on the following:
  - The number of allowable attachments to any given peer
  - The number of hops used for searching for a peer
  - Which peers know about which other peers
  - Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
- Weaknesses:
  - Managing security, data consistency, data/service availability, backup, and recovery are all more complex.
  - Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

JXTA Design Constraints

Interoperability

- software vendors *tend to* create specific code for their services e.g. file sharing, instant messaging etc
- incompatible systems
- duplicate effort
- JXTA attempts give peers a common language to talk to each other

Platform independence

JXTA technology is designed to be independent of:

- programming languages e.g. C or Java
- system platforms e.g. Microsoft Windows and UNIX
- networking platforms (such as TCP/IP or Bluetooth)

Ubiquity

- implementable on every device with a digital heartbeat e.g. PDAs, phones, sensors, consumer electronics appliances, network routers, desktop computers, data-center servers, storage systems
- avoid specific binding to platforms (Wintel…)
- future proof e.g. such technologies should be extended to new platforms e.g. mobile phones etc e.g. using J2ME

JXTA Current Implementations

JXTA Platform Current Implementations

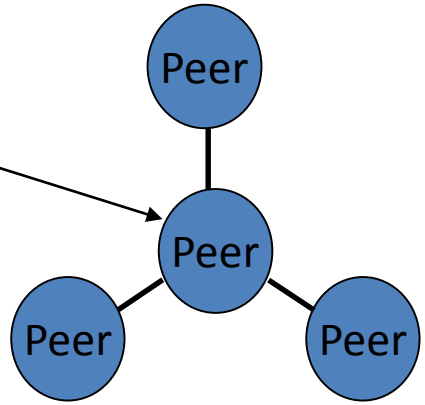
- JXTA for the Java 2 Platform Standard Edition (J2SE)—the reference implementation
- JXTA for the Java 2 Platform Micro Edition (J2ME)—for MIDP-1.0 compliant devices such as cell phones, PDAs, and controllers
- JXTA for PersonalJavaTM technology—for devices such as PDAs and webpads
- JXTA for C
- JXTA for PERL
- JXTA for Python
- JXTA for Ruby

JXTA Transport Current Implementations

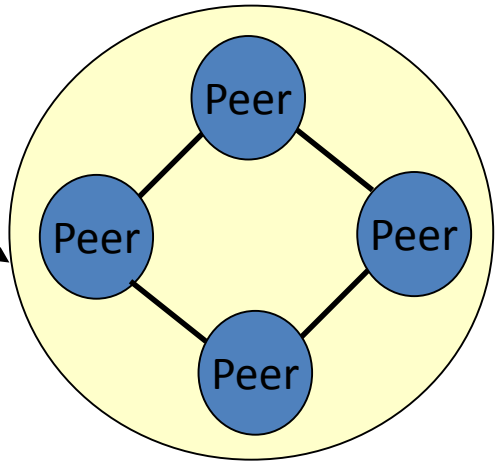
- TCP
- HTTP
- BEEP

JXTA Terms

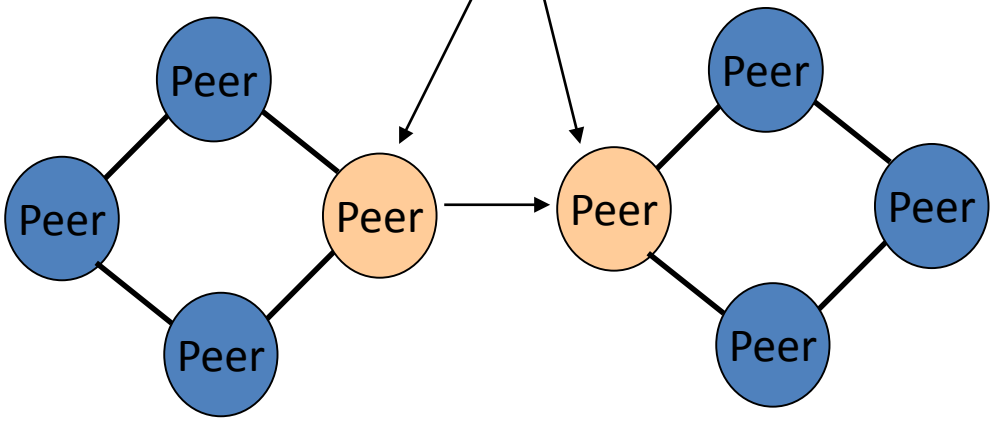
Peer: A JXTA node.



JXTA Group: a group is a set of JXTA nodes who share a common interest



A Rendezvous Peer: a meeting place e.g. gateway for JXTA groups



What is JXTA ?

6 protocols:

- Peer Discovery Protocol
- Peer Resolver Protocol
- Peer Information Protocol
- Pipe Binding Protocol
- Endpoint Routing Protocol
- Rendezvous Protocol

JXTA is a set of open, generalized P2P protocols that allow any connected device on the network to communicate and collaborate

JXTA is middleware – designed as a set of building blocks to allow developers to rapidly develop P2P applications

JXTA is designed to have a peer-to-peer, decentralized model (although JXTA supports traditional client/centralized server and brokered)

As in Gnutella, every JXTA peer can be both a client and a server

JXTA Overview

Project JXTA defines a set of **six protocols** , which allow peers to:

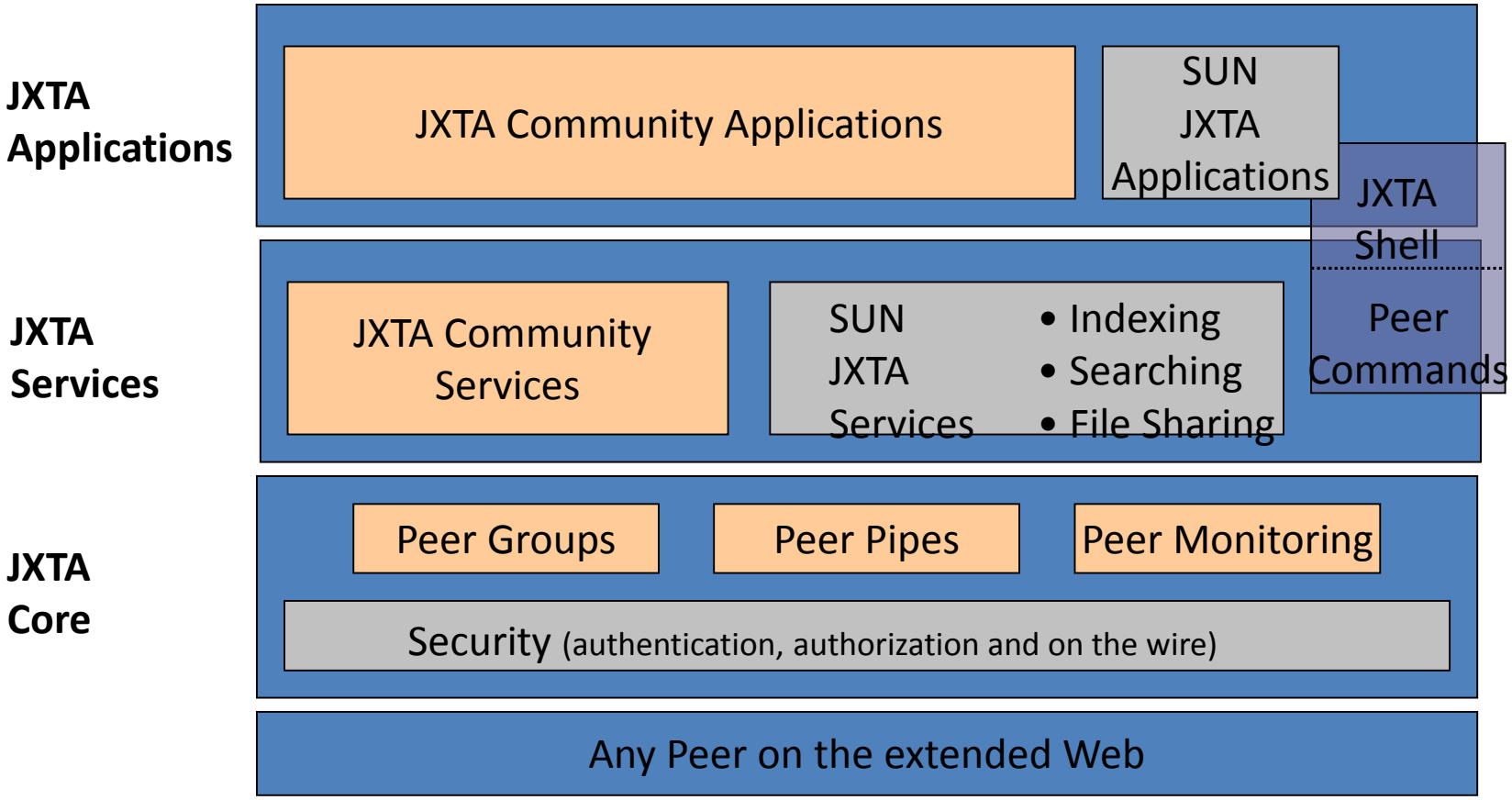
- Discover each other
- Self-organize into peer groups
- Advertise and discover network services
- Communicate with each other
- Monitor each other

…and the protocols **do not** require the use of any particular:

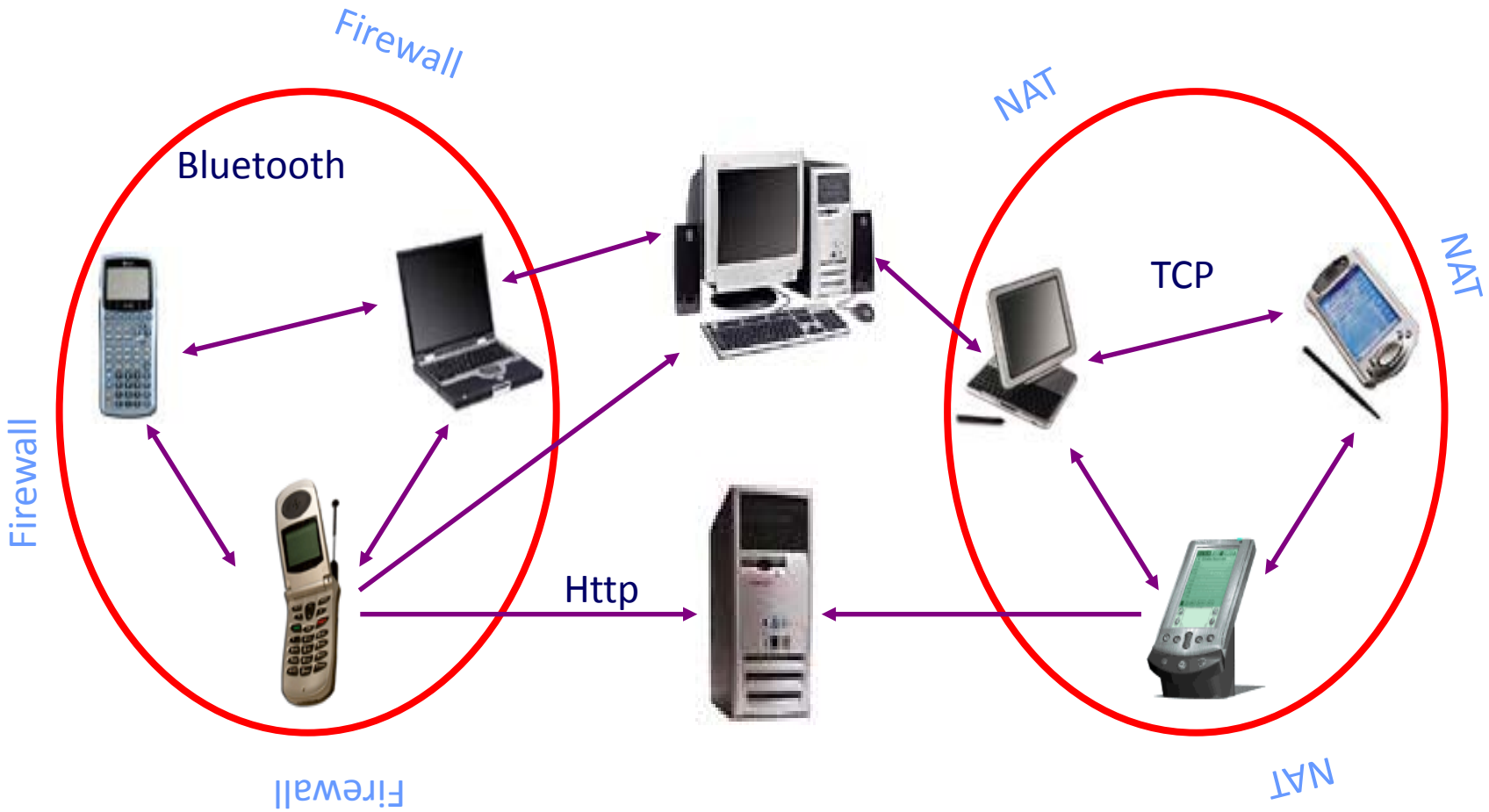
- programming language
- operating system
- network transport
- Topology
- authentication
- Security
- encryption model.

The JXTA protocols therefore allow heterogeneous devices with completely different software stacks to interoperate.

JXTA Architecture



Devices in JXTA Network



- a distributed decentralized set of heterogeneous devices

JXTA Terms and Concepts

Peer: any networked device that implements one or more of the JXTA protocols

Advertisements: XML structured document that names, describes, and publishes the existence of a resource e.g. peer, peer group, pipe, or service.

Messages: sent between peers can be XML or binary

Pipes: messages are send through virtual pipes – see next section …

Identifiers: each JXTA entity e.g. peer, advert etc has a UUID identifier

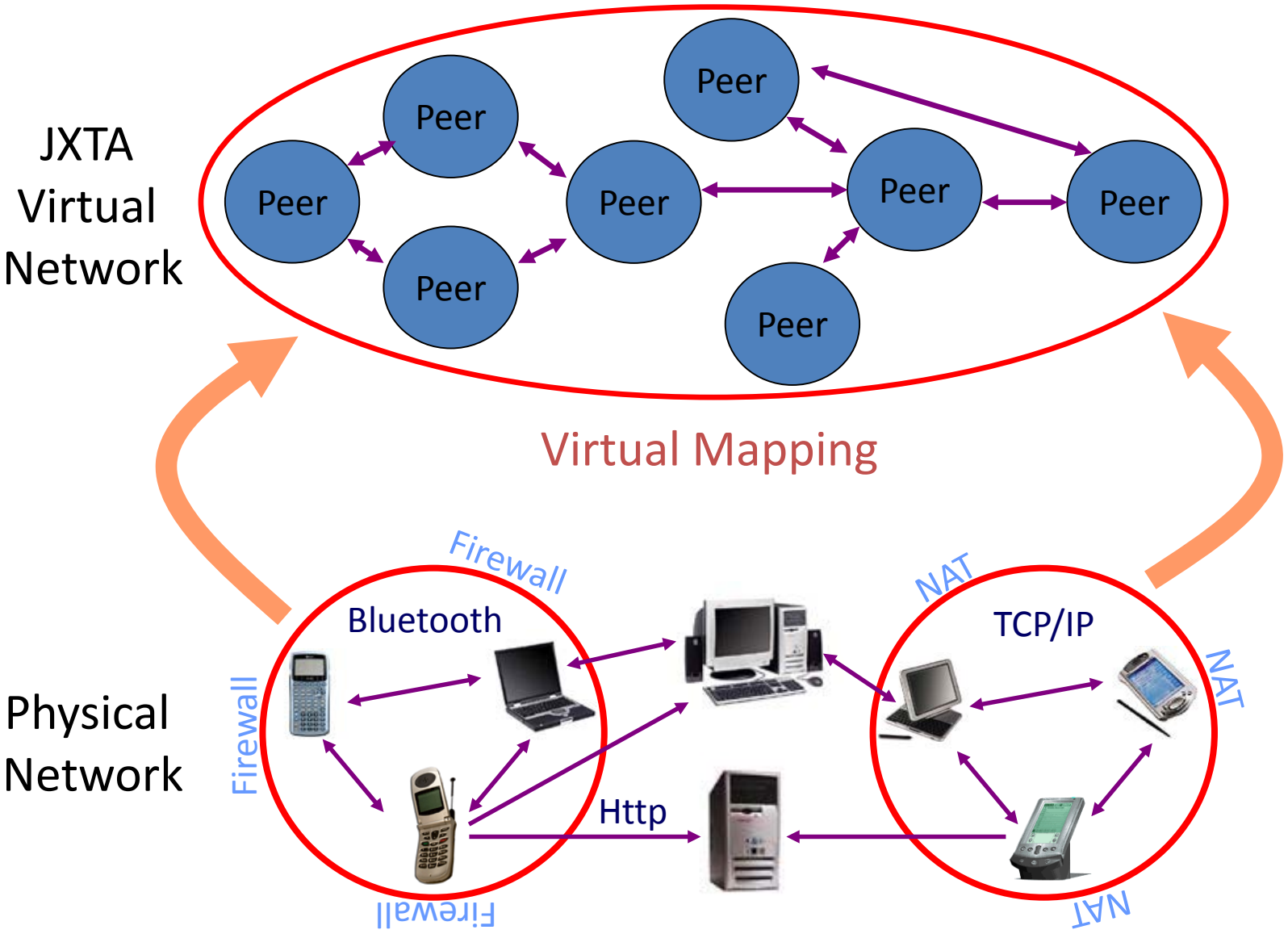
Rendezvous Nodes: a caching nodes for advertisements – similar to the super/ reflector nodes in lecture 4.

Relay Nodes: JXTA routers – help to route messages via firewalls, NAT systems etc – i.e. they relay the message on

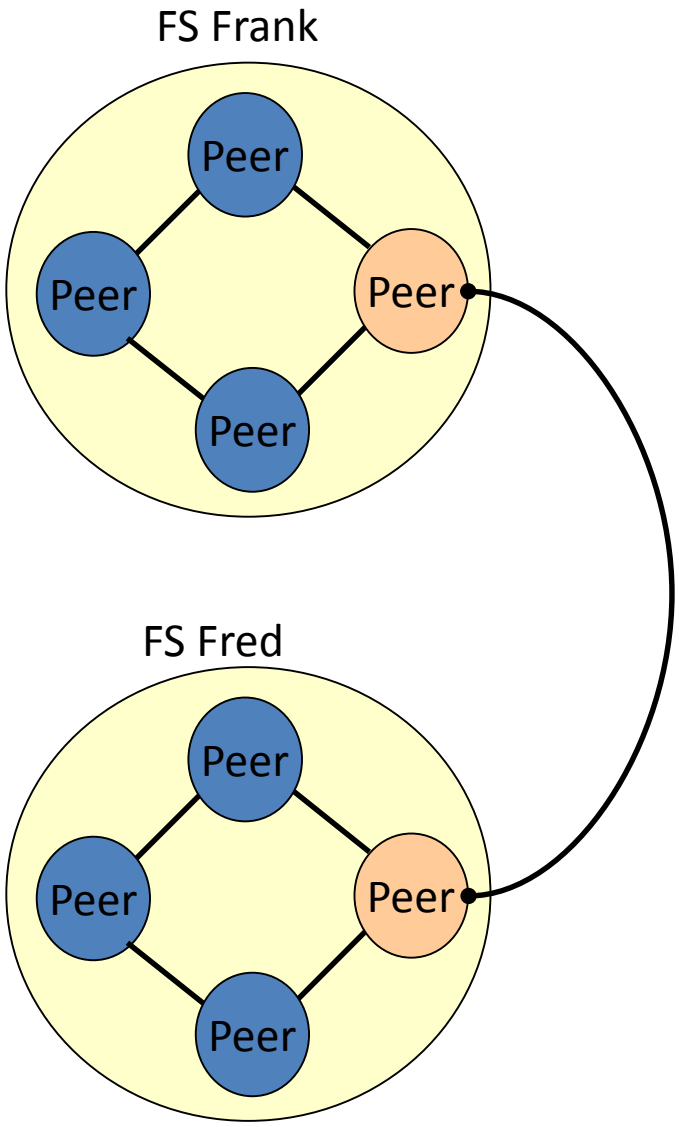
Virtual JXTA

- JXTA Virtual Network overlay
- JXTA Groups
- JXTA Virtual Pipes

JXTA Virtual Mapping



JXTA Groups



- virtual entity - speak a set of peer group protocols

- collection of cooperating peers providing a common set of services e.g. file sharing peer group, a CPU sharing peer group.

- Peer group boundaries define search scope

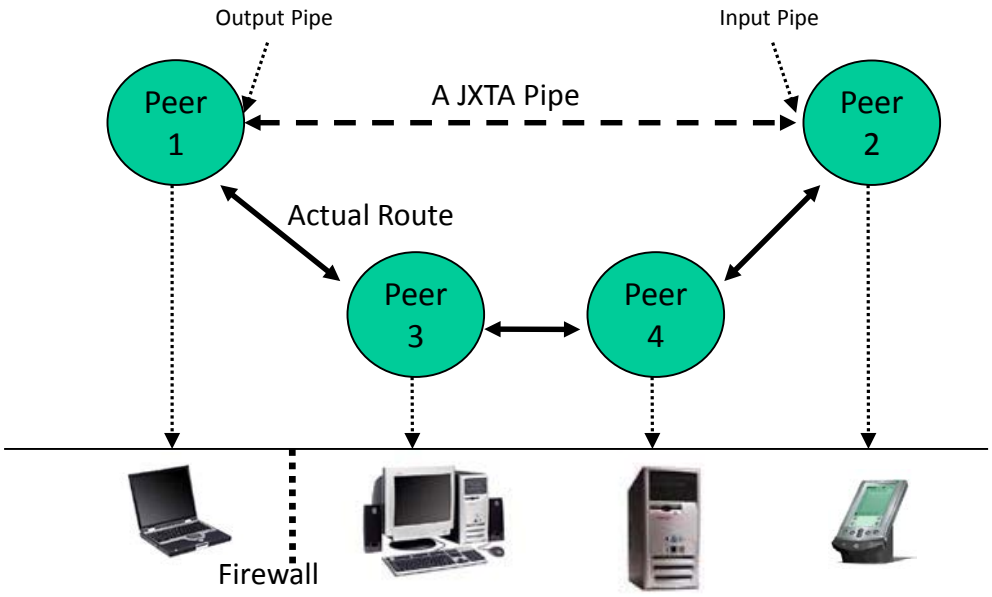
- can be used to create a monitoring environment

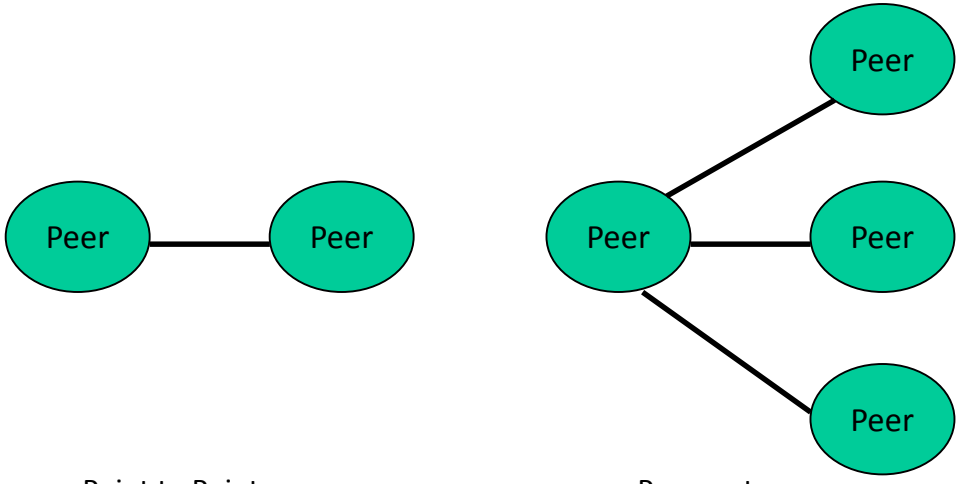
- can be password protected and implement local security policies

- one special group, called the *World Peer Group* (the default peer group a peer joins) that includes all JXTA peers.

- At least one rendezvous for a group – groups are the scoping environment for a rendezvous

JXTA Pipes



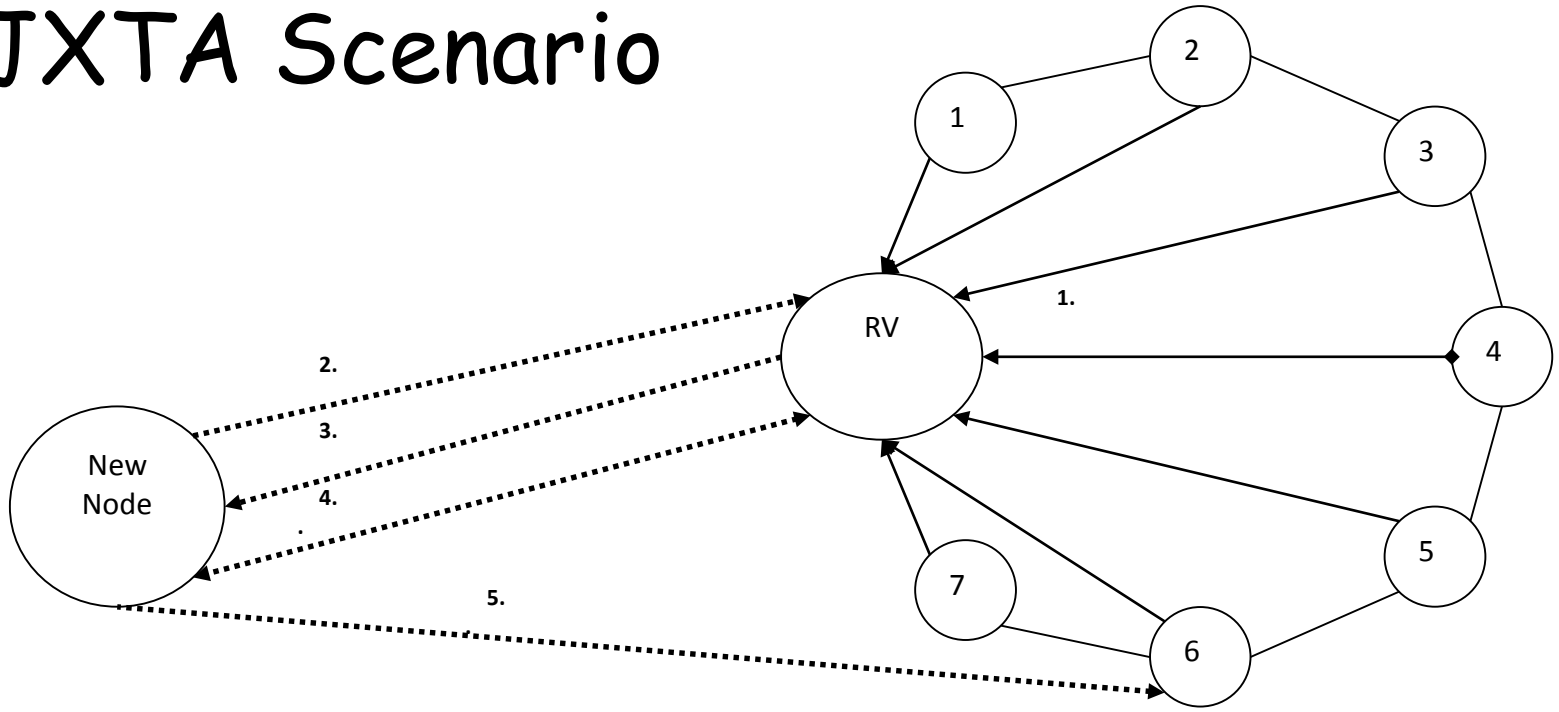


Point to Point

Propagate

- Virtual Communication Channels
  - Switchable e.g. TCP/IP, HTTP, Bluetooth
  - NOT point to point
- Pipe endpoints -> *input pipe* (the receiving end) and the *output pipe* (the sending end).
- Asynchronous and unidirectional
- Messages flow from the output pipe into the input pipes.
- Support the transfer of any object, including binary code, data strings, and Java technology-based objects
- Two Types:
  - (End) Point to (End) Point
  - Propagate - multicast

JXTA Scenario

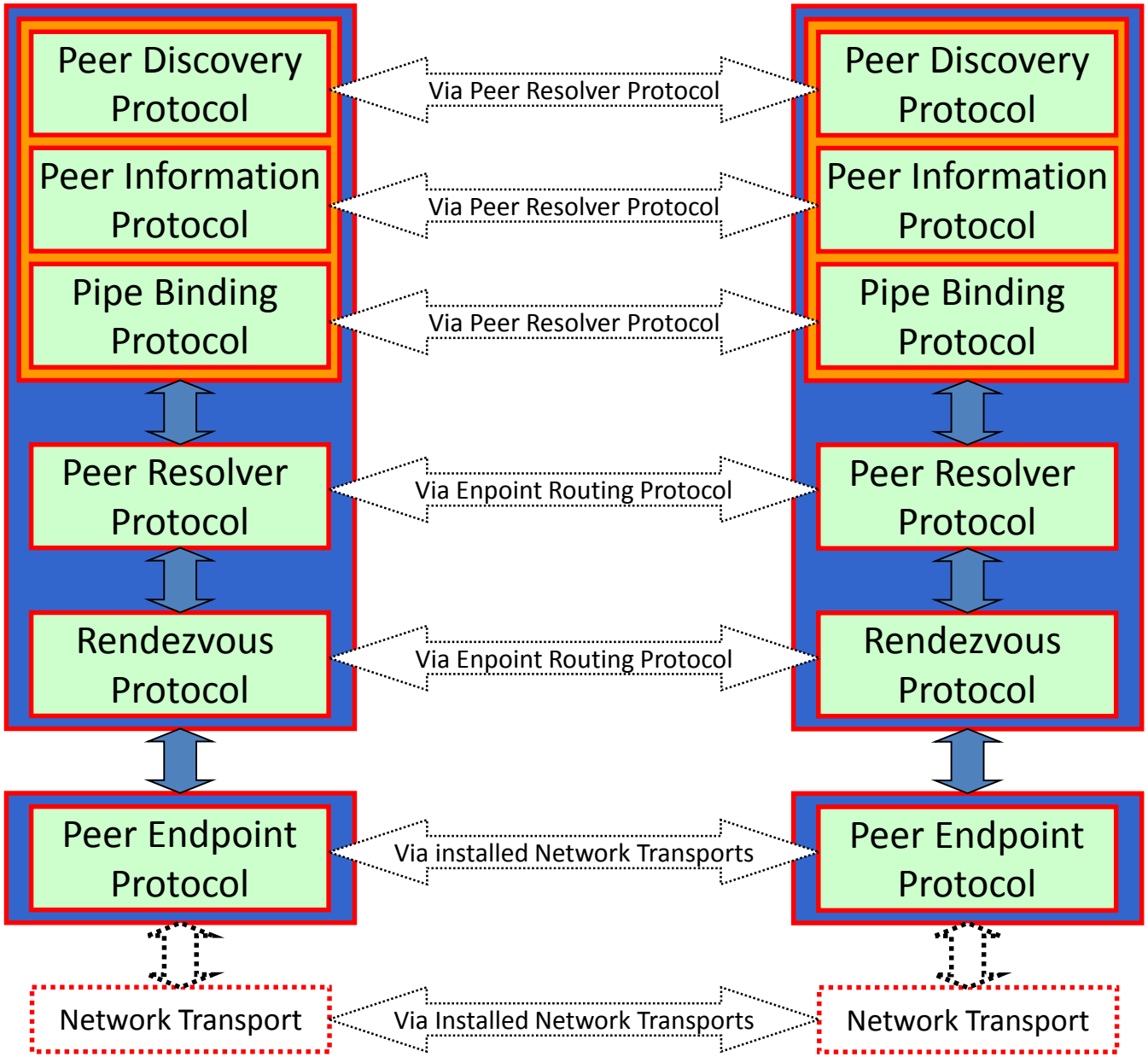


1. Rendezvous node (RV) accepts connection for nodes 1-7 and stores advertisements locally
2. New node contacts Rendezvous using a discovery mechanism e.g. Unicast/multicast (PDP)
3. RV authenticates New Node and adds the New Node to the group (RVP)
4. New Nodes performs a file search query by contacting the RV find a match locally or propagates this query to all other members in the group. The file is found on node 6 (PDP)
5. New Node and node 6 communicate directly through a JXTA pipe. This connection is virtual and may actually traverse (route) through the RV node and node 7.

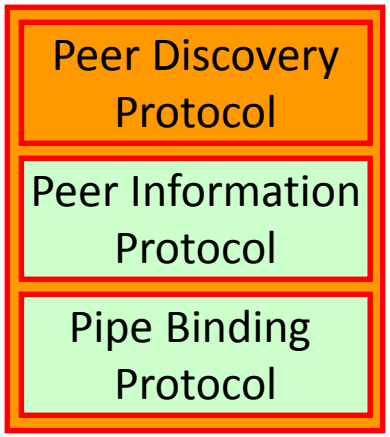
JXTA Protocols

- Peer Discovery Protocol
- Peer Information Protocol
- Pipe Binding Protocol
- Peer Resolver Protocol
- Rendezvous Protocol
- Peer Endpoint Protocol

JXTA Protocol Stack



Peer Discovery

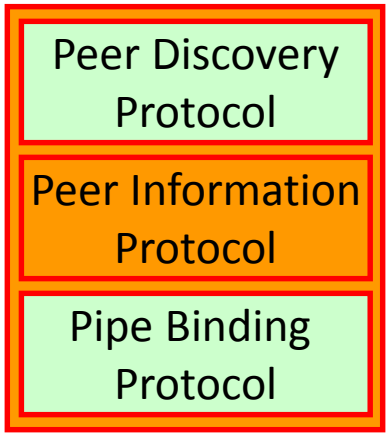


- A peer uses the PDP to discover a JXTA resource
- resources are described by advertisements e.g. can be services, pipes, peers, peer groups, or any other advertisements
- Note, that the first word, peer, is the subject and not necessarily the object
- Using this protocol, peers can advertise their own resources, and discover the resources from other peers
- Peer resources are published using XML-based advertisements

Two levels of discovery:

1. Joining a JXTA network
    1. Multicast
    2. Unicast
2. discovering JXTA resource within a JXTA network.

Finding Information about Peers



- allows peers to learn about the capabilities and status of other peers e.g. uptime, traffic load, capabilities, state etc
  - e.g. one can send a *ping* message to see if a peer is alive.
- also query a peer's properties where each property as a name and a value string
- useful for implementing monitoring

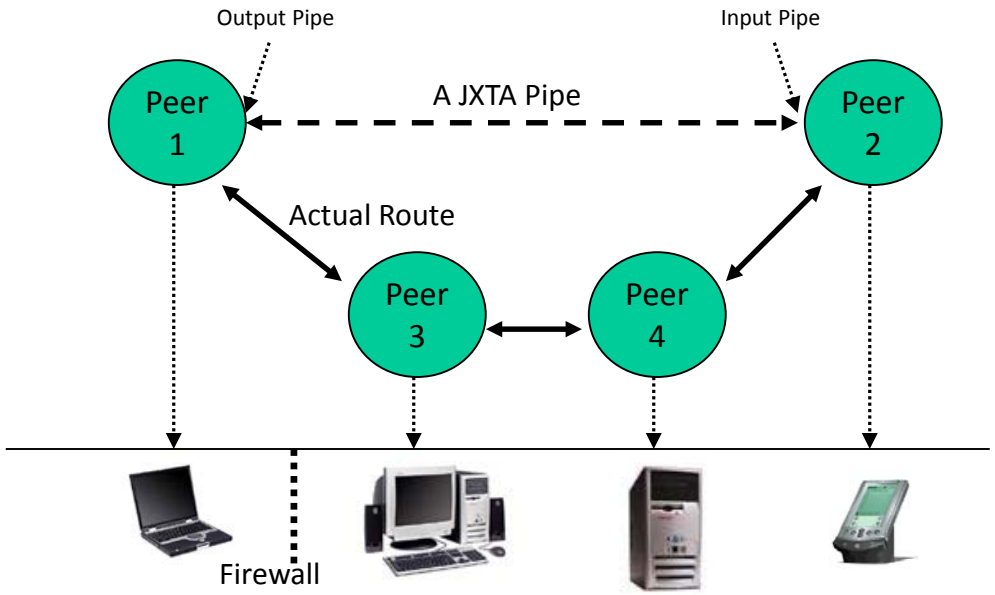
Binding Pipes

Peer Discovery Protocol

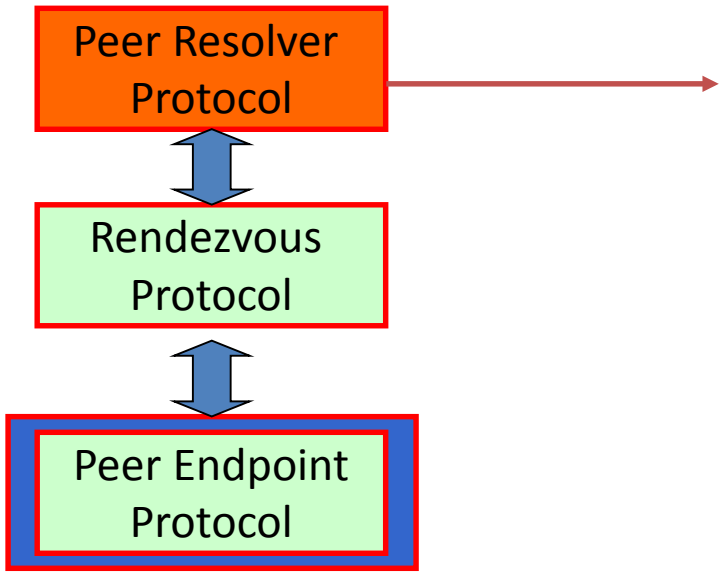
Peer Information Protocol

Pipe Binding Protocol

- allows a peer to establish a virtual communication channel (i.e. a pipe) between peers
- allows the binding of the two or more ends of the pipe endpoints forming the connection
- a peer binds a pipe advertisement to a pipe endpoint thus indicating here messages actually go over the pipe
- Bind occurs during the open operation, whereas unbind occurs during the close operation.

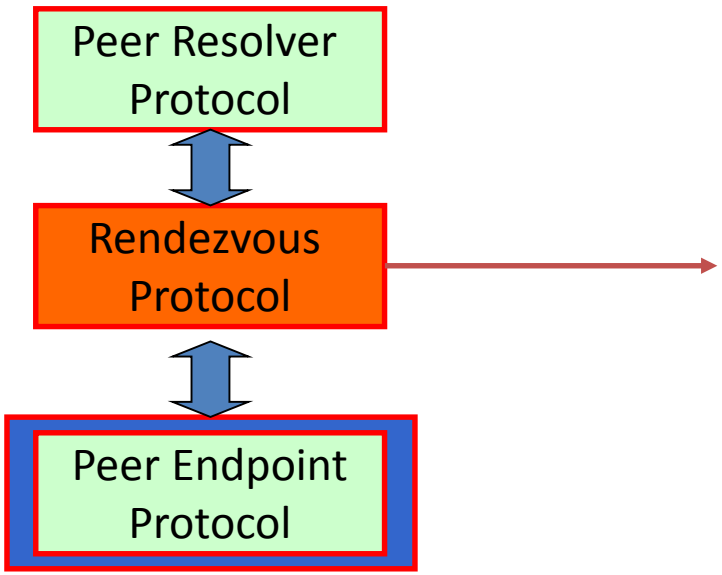


'The' Resolver



- enables a peer to implement high-level search capabilities
- allows a peer to send and receive generic queries to find or search for peers, peer groups, pipes, and other information

Rendezvous



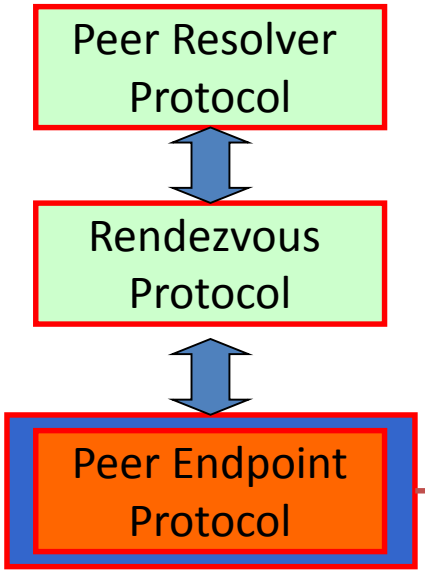
- allows a Peer to send messages to all the listeners of the service

- The rendezvous protocol defines how a peer can subscribe or be a subscriber to a propagation service allowing larger communities to form

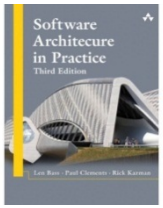
- A rendezvous nodes' scope is a peer group

- e.g. the rendezvous protocol is used by the peer resolver protocol and by the pipe binding protocol in order to propagate messages.

Routing Those Messages



- allows a peer to find information about the available routes for sending a message to destination peer
- i.e. pipes are often not directly connected to each other
- allows the implementation of routing algorithms into JXTA
- Peers implementing the endpoint routing protocol respond to queries with available route information giving a list of gateways along the route.

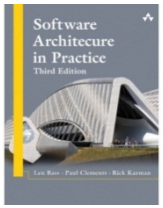


Exercises

- Give an example of P2P pattern
- Use P2P pattern to design the following cases

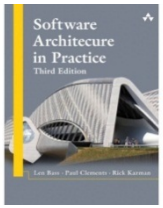
?

?

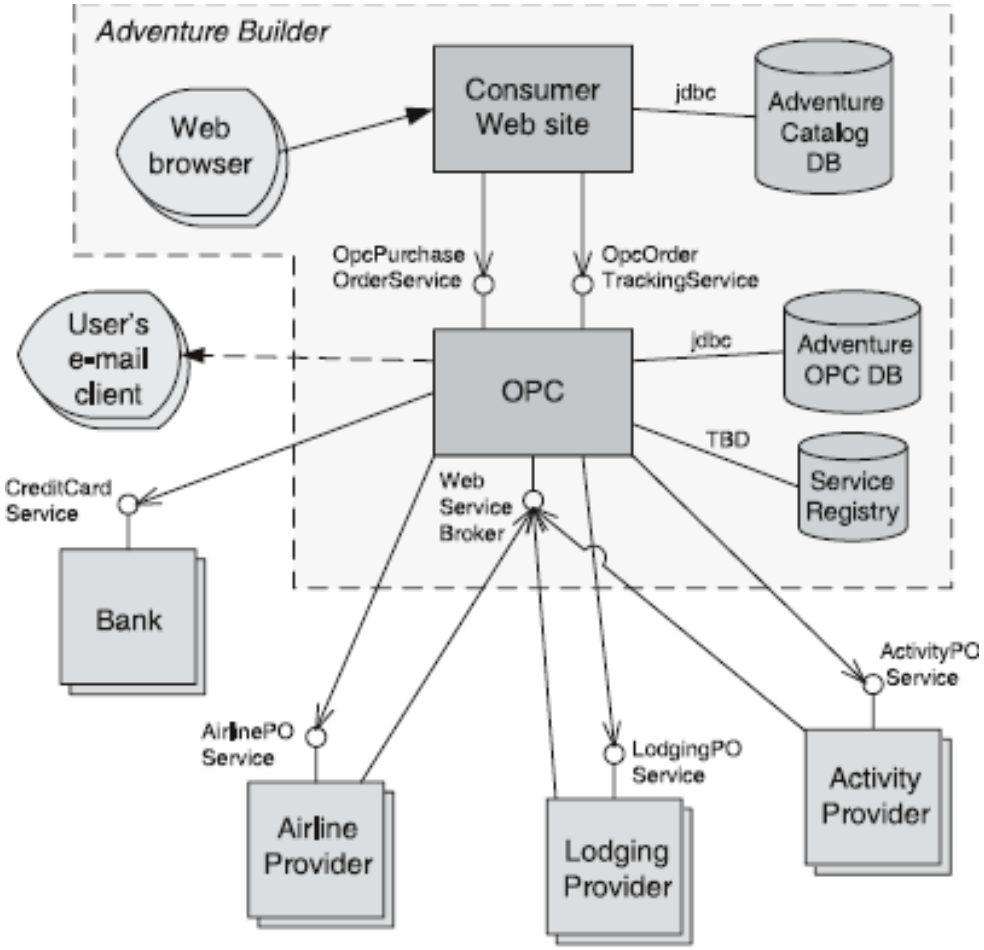


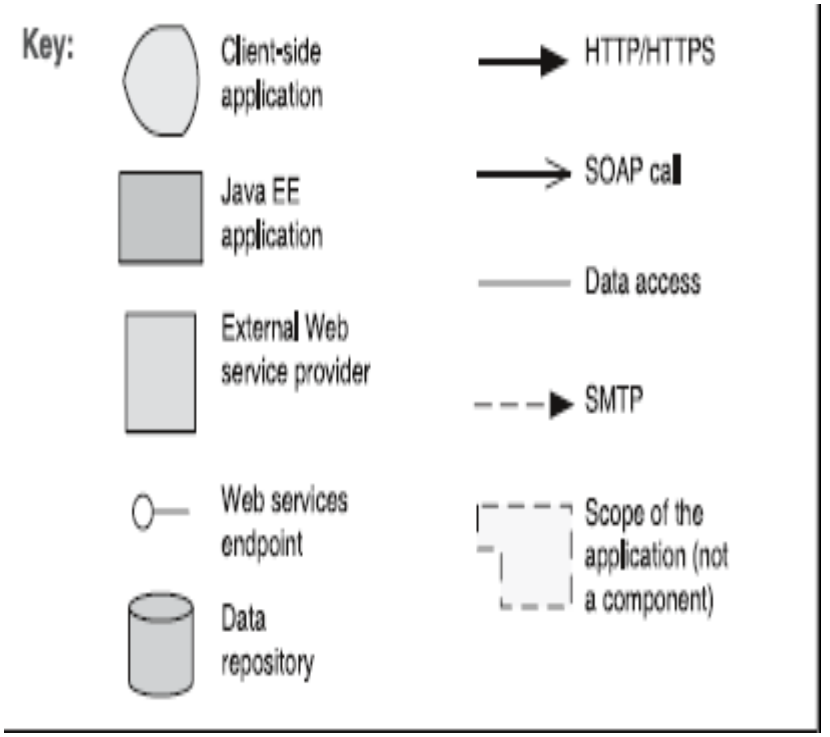
Service Oriented Architecture Pattern

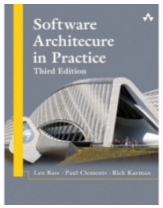
- **Context**: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.
- **Problem**: How can we support **interoperability** of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?
- **Solution**: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services.



Service Oriented Architecture Example

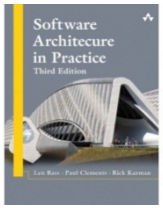






Service Oriented Architecture Solution - 1

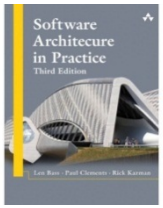
- Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network.
- Elements:
  - Components:
    - *Service providers,* which provide one or more services through published interfaces.
    - *Service consumers,* which invoke services directly or through an intermediary.
    - *Service providers* may also be service consumers.
  - *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers.
  - *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime.
  - *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows.



Service Oriented Architecture Solution - 2

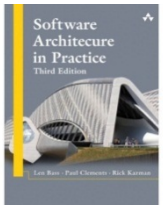
– Connectors:

- *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP.
- *REST connector,* which relies on the basic request/reply operations of the HTTP protocol.
- *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.



Service Oriented Architecture Solution - 3

- Relations: Attachment of the different kinds of components available to the respective connectors
- Constraints: Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
- Weaknesses:
  - SOA-based systems are typically complex to build.
  - You don’t control the evolution of independent services.
  - There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

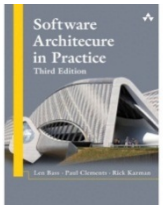


Exercises

- Give an example of SOA pattern
- Use SOA pattern to design the following cases

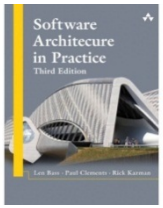
?

?

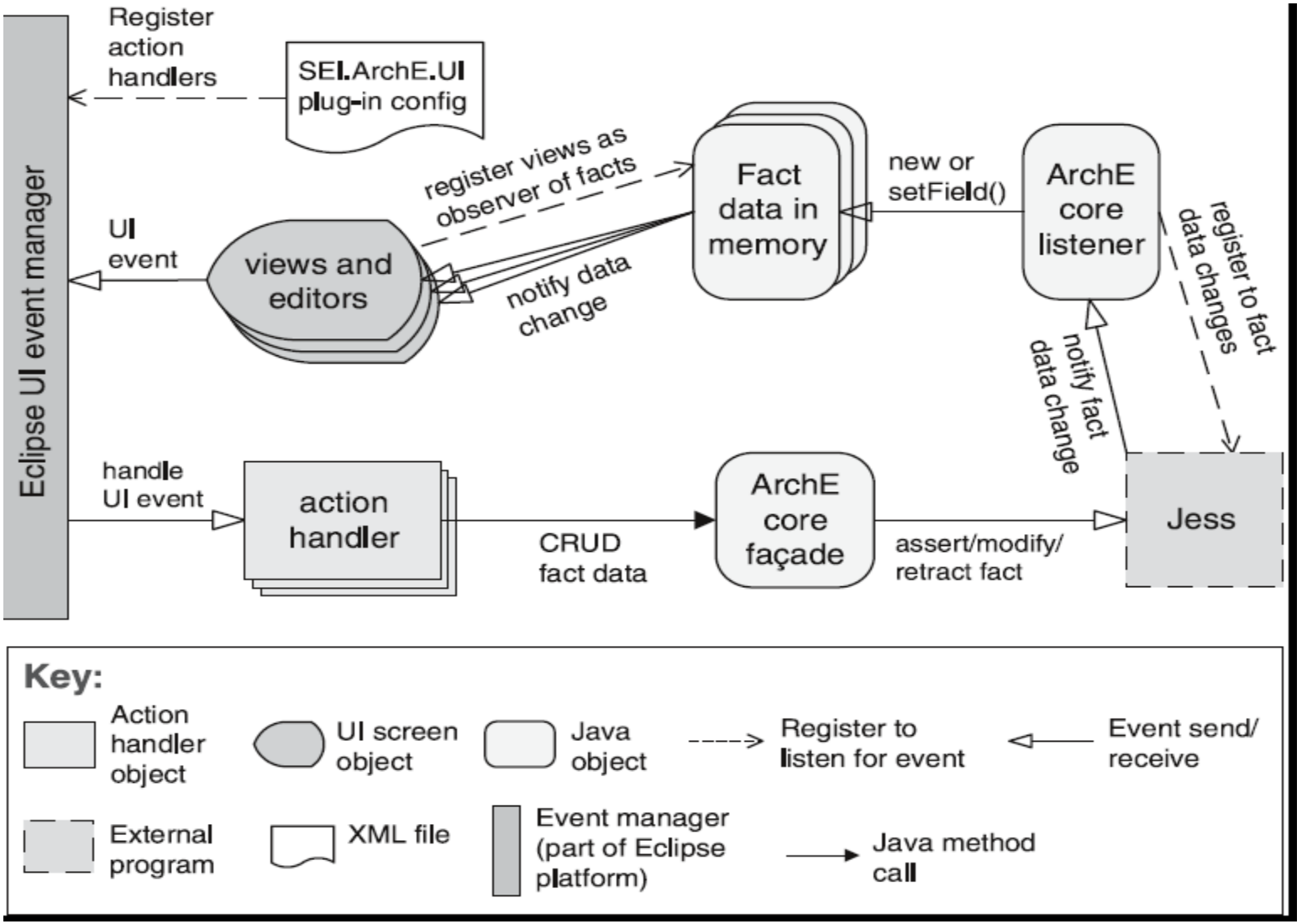


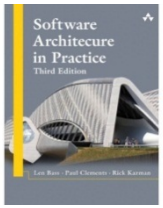
Publish-Subscribe Pattern

- **Context:** There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.
- **Problem:** How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?
- **Solution:** In the publish-subscribe pattern, components interact via announced messages, or events. Components may subscribe to a set of events. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events.



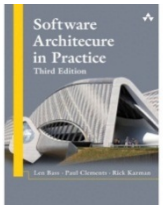
Publish-Subscribe Example





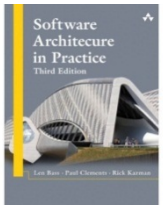
Publish-Subscribe Solution – 1

- Overview: Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
- Elements:
  - *Any C&C component* with at least one publish or subscribe port.
  - *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
- Relations: The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.



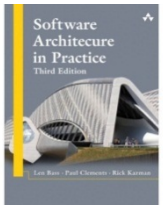
Publish-Subscribe Solution - 2

- Constraints: All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles.
- Weaknesses:
  - Typically increases latency and has a negative effect on scalability and predictability of message delivery time.
  - Less control over ordering of messages, and delivery of messages is not guaranteed.



Example: Google's Guava EventBus

```
// Class is typically registered by the container.
class EventBusChangeRecorder {
  @Subscribe public void recordCustomerChange(ChangeEvent e) {
    recordChange(e.getChange());
  }
}
// somewhere during initialization
eventBus.register(new EventBusChangeRecorder());
// much later
public void changeCustomer() {
  ChangeEvent event = getChangeEvent();
  eventBus.post(event);
}
```

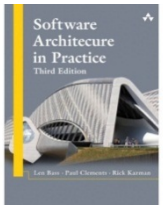


Exercises

- Give an example of Publish&Subscribe pattern
- Use Publish&Subscribe pattern to design the following cases

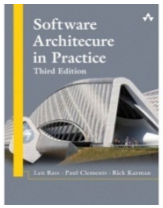
?

?



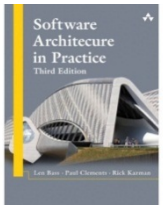
Observer Vs. Pub/Sub

- Observer pattern is mostly implemented in a synchronous way, i.e. the observable calls the appropriate method of all its observers when some event occurs. The Pub/Sub pattern is mostly implemented in an asynchronous way (using message queue).
- Also, in the observer pattern, the observers are aware of the observable. Whereas, in Pub/Sub, neither the publishers, nor the consumers need to know each other. They simply communicate with the help of message queues.

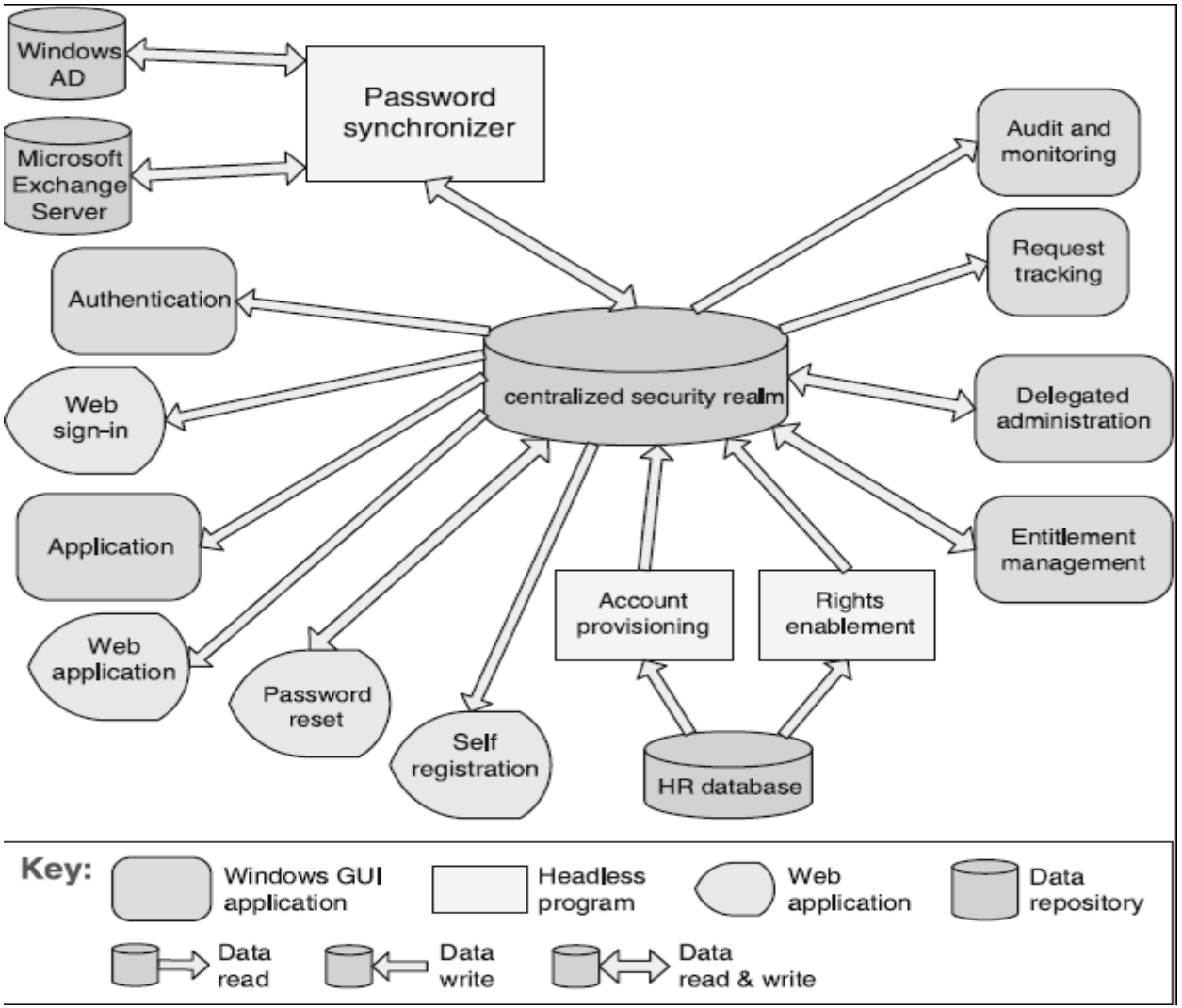


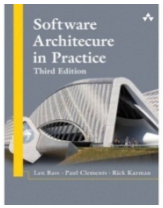
Shared-Data Pattern

- **Context:** Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.
- **Problem:** How can systems store and manipulate persistent data that is accessed by multiple independent components?
- **Solution:** In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*.



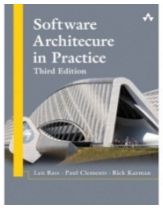
Shared Data Example





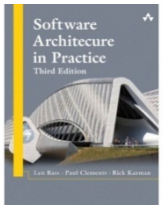
Shared Data Solution - 1

- Overview: Communication between data accessors is mediated by a shared data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.
- Elements:
  - *Shared-data store.* Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted.
  - *Data accessor component*.
  - *Data reading and writing connector*.



Shared Data Solution - 2

- Relations: *Attachment* relation determines which data accessors are connected to which data stores.
- Constraints: Data accessors interact only with the data store(s).
- Weaknesses:
  - The shared-data store may be a performance bottleneck.
  - The shared-data store may be a single point of failure.
  - Producers and consumers of data may be tightly coupled.

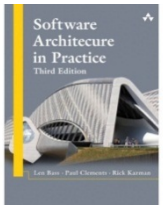


Exercises

- Give an example of Shared Data pattern
- Use Shared Data pattern to design the following cases

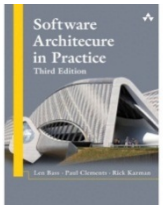
?

?

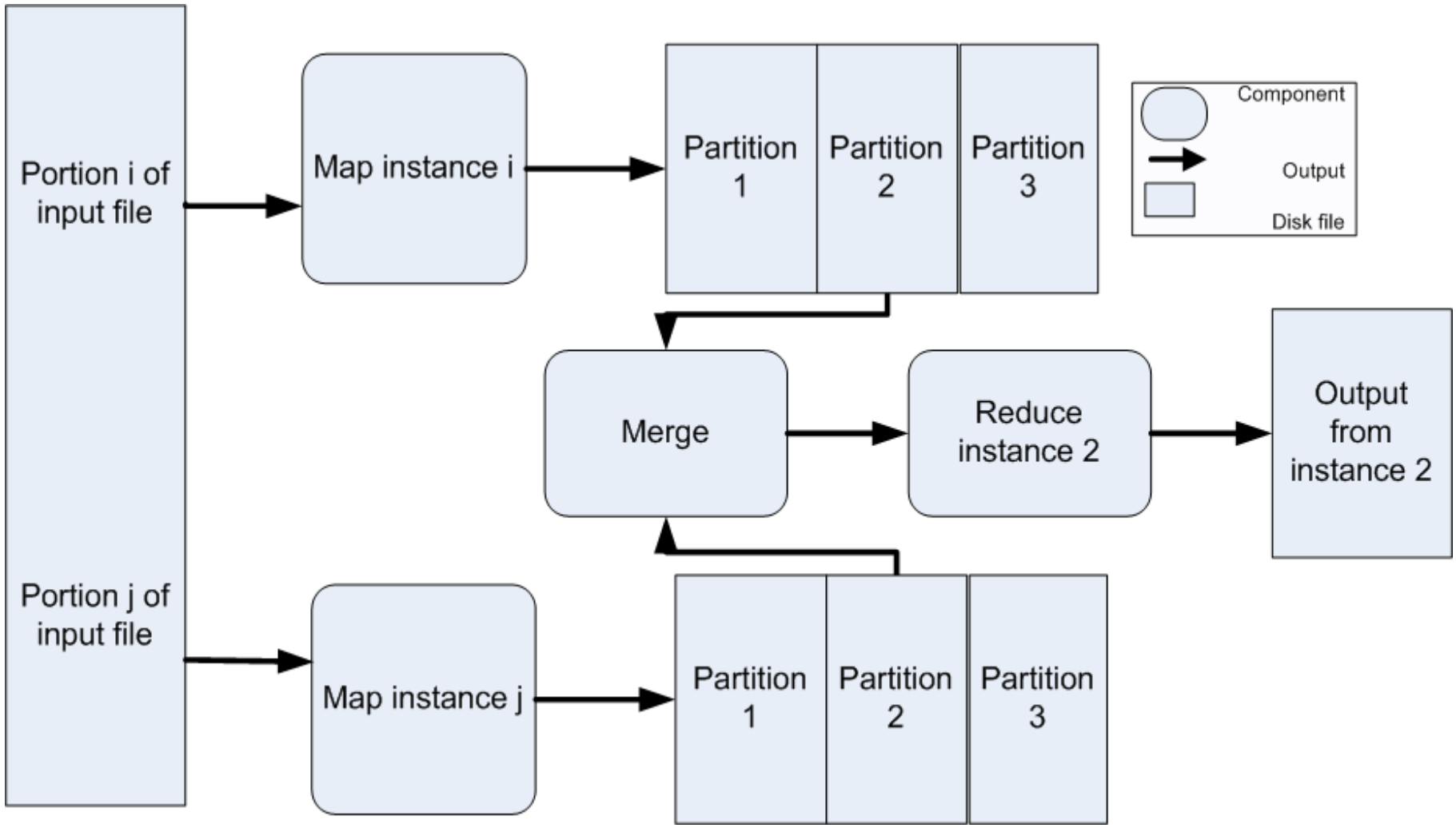


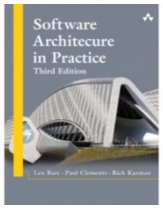
Map-Reduce Pattern

- **Context:** Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale.
- **Problem:** For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.
- **Solution:** The map-reduce pattern requires three parts:
  - A specialized infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and handles sorting the data as needed.
  - A programmer specified component called the map which filters the data to retrieve those items to be combined.
  - A programmer specified component called reduce which combines the results of the map



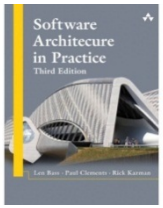
Map-Reduce Example





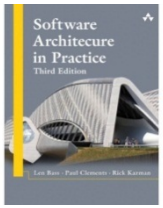
Map-Reduce Solution - 1

- Overview: The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the extract and transform portions of the analysis and the reduce performs the loading of the results.
- Elements:
  - Map is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis.
  - Reduce is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load.
  - The infrastructure is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.



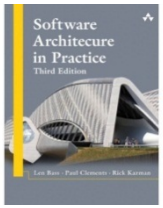
Map-Reduce Solution - 2

- Relations:
  - Deploy on is the relation between an instance of a map or reduce function and the processor onto which it is installed.
  - Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce.
- Constraints:
  - The data to be analyzed must exist as a set of files.
  - Map functions are stateless and do not communicate with each other.
  - The only communication between map reduce instances is the data emitted from the map instances as <key, value> pairs.
- Weaknesses:
  - If you do not have large data sets, the overhead of map-reduce is not justified.
  - If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.
  - Operations that require multiple reduces are complex to orchestrate.

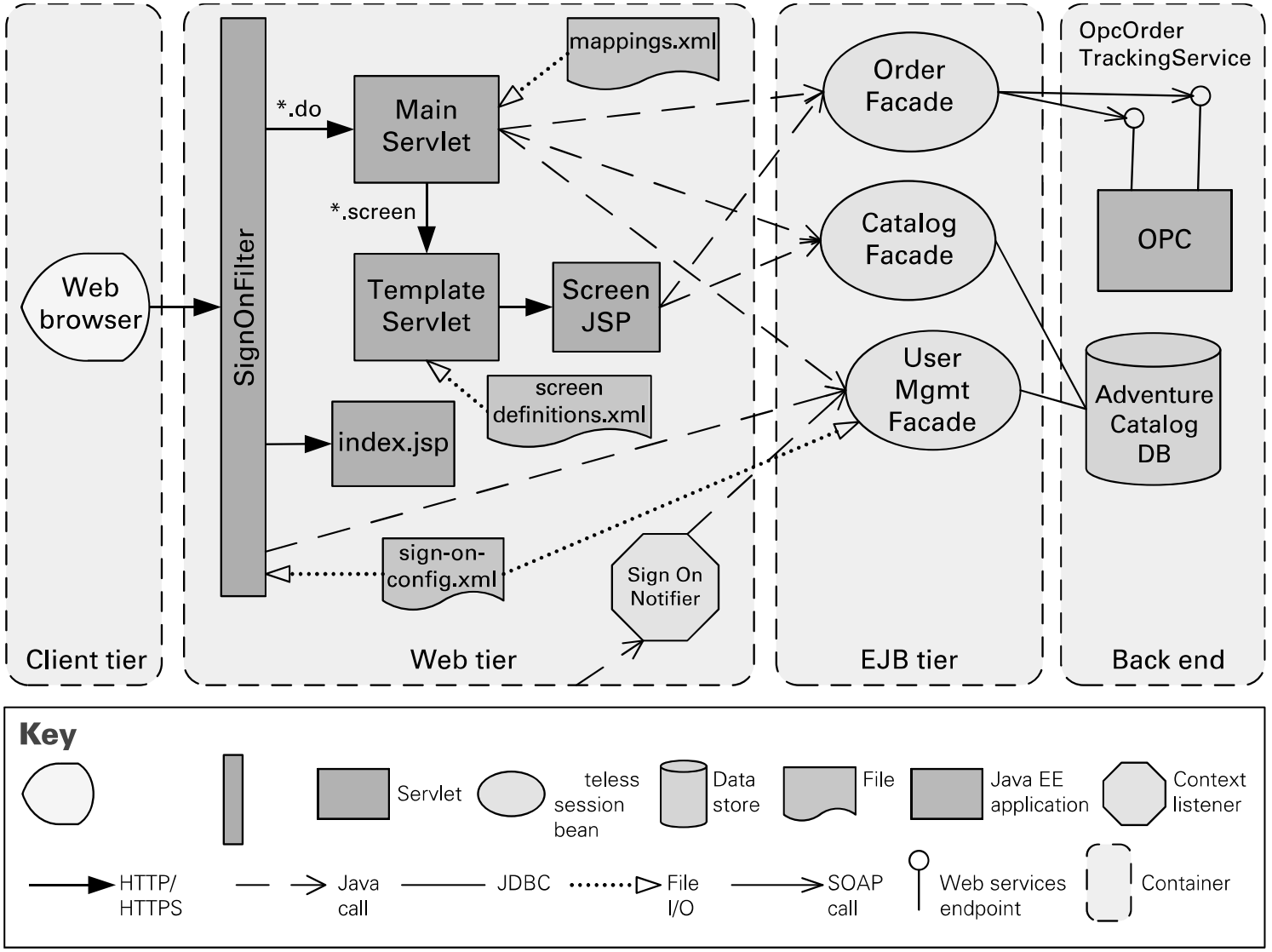


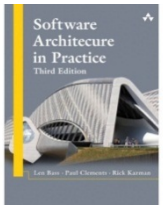
Multi-Tier Pattern

- **Context**: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets.
- **Problem**: How can we split the system into a number of computationally independent execution structures—groups of software and hardware—connected by some communications media?
- **Solution**: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a tier.



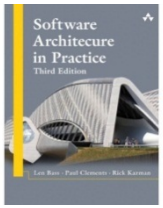
Multi-Tier Example





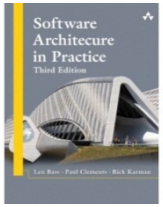
Multi-Tier Solution

- Overview: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*.
- Elements:
  - *Tier,* which is a logical grouping of software components.
- Relations:
  - *Is part of,* to group components into tiers.
  - *Communicates with,* to show how tiers and the components they contain interact with each other.
  - *Allocated to,* in the case that tiers map to computing platforms.
- Constraints: A software component belongs to exactly one tier.
- Weaknesses: Substantial up-front cost and complexity.



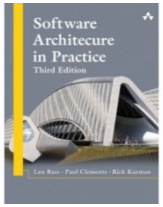
Relationships Between Tactics and Patterns

- Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom.
- MVC, for example utilizes the tactics:
  - Increase semantic coherence
  - Encapsulation
  - Use an intermediary
  - Use run time binding



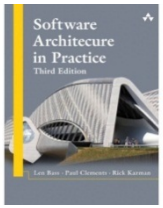
七种构建模式的比较

构建模式	主要特点	主要优点	主要缺点	适合领域	说明
管道-过滤器风格	过滤器相对独立	功能模块复用；强可维护性和可扩展性；具有并发性；模块独立性高	不适于交互性强的应用；对于存在关系的数据流必须进行协调	系统可划分清晰的模块；模块相对独立；有清晰的模块接口	每个功能模块有一组输入输出，模块划分限制较大。
面向对象风格	力取实现问题空间和软件系统空间结构的一致性	高度模块性；实现封装；代码共享；灵活；易维护；可扩充性好	增加了对象之间的依赖关系	多种领域	是现在使用非常多的一种构建模式
事件驱动风格	系统由若干子系统构成且成为一个整体；系统有统一的目标；子系统有主从之分；每一子系统有自己的事件收集和处理机制	适合描写系统组；容易实现并发处理和多任务；可扩展性好；具有类层次结构；简化代码；	因为树型结构所以削弱了对系统计算的控制能力；各个对象的逻辑关系复杂	一个系统对外部的表现可以从它对事件的处理表征出来	事件驱动系统具有某种意义上的帝归性，形成了“部分–整体”的层次结构



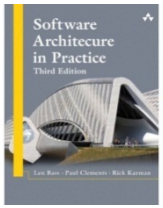
七种构建模式的比较

构建模式	主要特点	主要优点	主要缺点	适合领域	说明
分层风格	各个层次的组件形成不同功能级别的虚拟机；多层相互协同工作，而且实现透明	支持系统设计过程中的逐级抽象；可扩展性好；支持软件复用	不同层次之间耦合度高的系统很难实现	适合功能层次的抽象和相互之间低耦合的系统	
数据共享风格	采用两个常用构件中央数据单元和一些相对独立的组件集合	中央数据单元实现了数据的集中，以数据为中心	适合于特定领域	适合于专家系统等人工智能领域问题的求解	数据和处理功能分界明显，
解释器风格	系统核心是虚拟机	可以用多种操作来解释一个句子	适合于特定领域	适合于模式匹配系统和语言编译器	
反馈控制环风格	通过不断地测量被控对象，认识和掌控被控对象；将控制理论引入体系结构构建	将控制理论引入到计算机软件体系结构中	适合于特定领域	该系统中一定存在有目标的作用、信息处理、闭环和开环控制过程	我认为这种构建模式应用范围很小



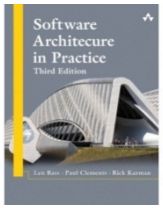
Tactics Augment Patterns

- Patterns solve a specific problem but are neutral or have weaknesses with respect to other qualities.
- Consider the broker pattern
  - May have performance bottlenecks
  - May have a single point of failure
- Using tactics such as
  - Increase resources will help performance
  - Maintain multiple copies will help availability



Tactics and Interactions

- Each tactic has pluses (its reason for being) and minuses – side effects.
- Use of tactics can help alleviate the minuses.
- But nothing is free…

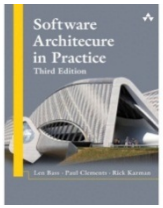


Tactics and Interactions - 2

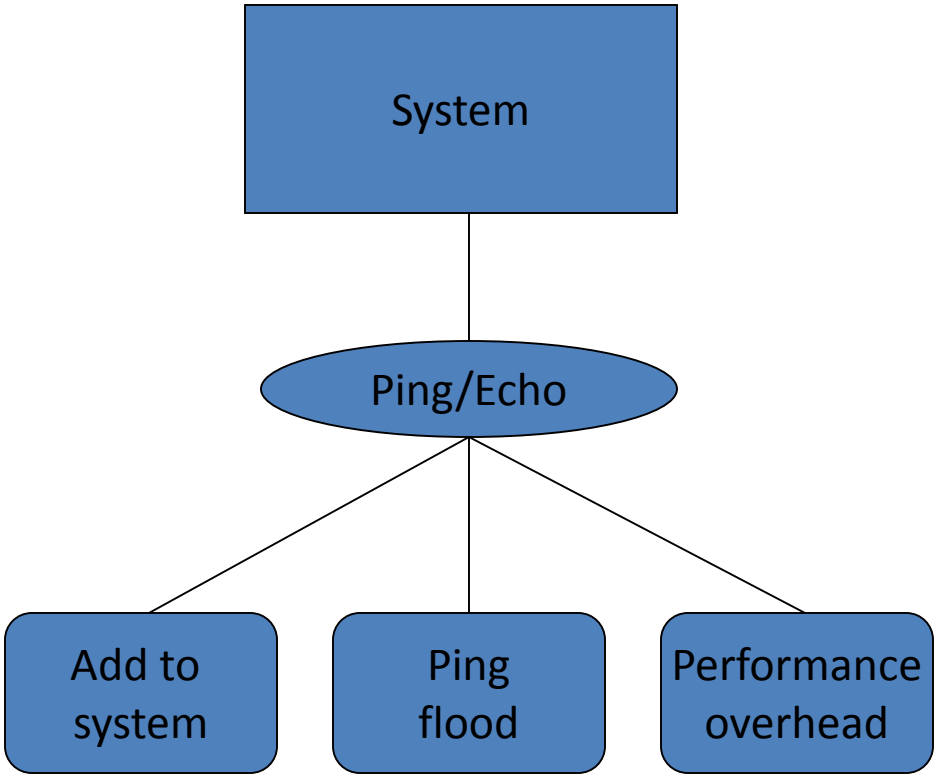
A common tactic for detecting faults is Ping/Echo.

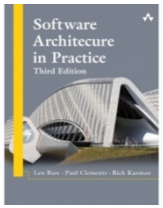
Common side-effects of Ping/Echo are:

- security: how to prevent a ping flood attack?
- performance: how to ensure that the performance overhead of ping/echo is small?
- modifiability: how to add ping/echo to the existing architecture?



Tactics and Interactions - 3



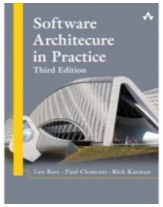


Tactics and Interactions - 4

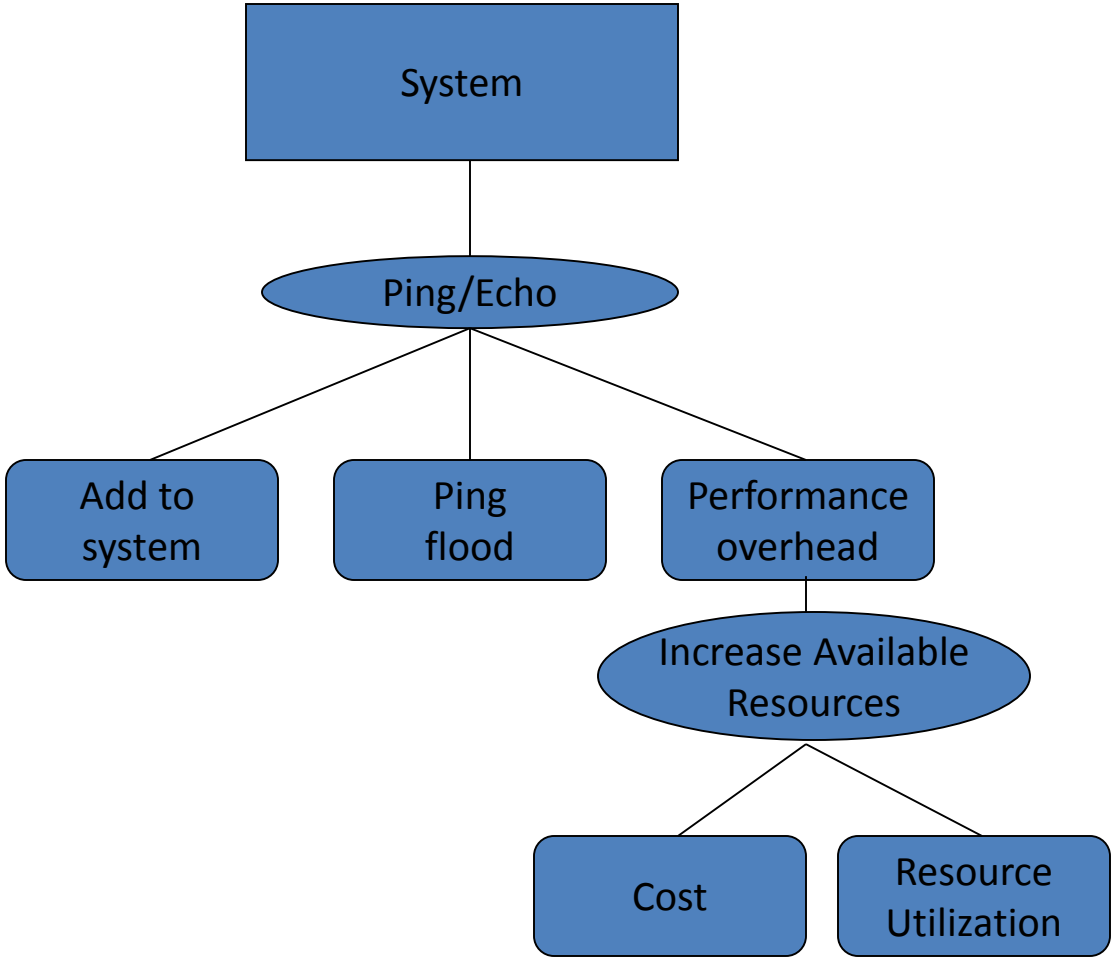
A tactic to address the performance side-effect is “Increase Available Resources”.

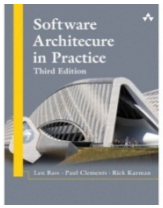
Common side effects of Increase Available Resources are:

- cost: increased resources cost more
- performance: how to utilize the increase resources efficiently?



Tactics and Interactions - 5



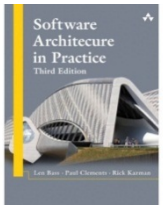


Tactics and Interactions - 6

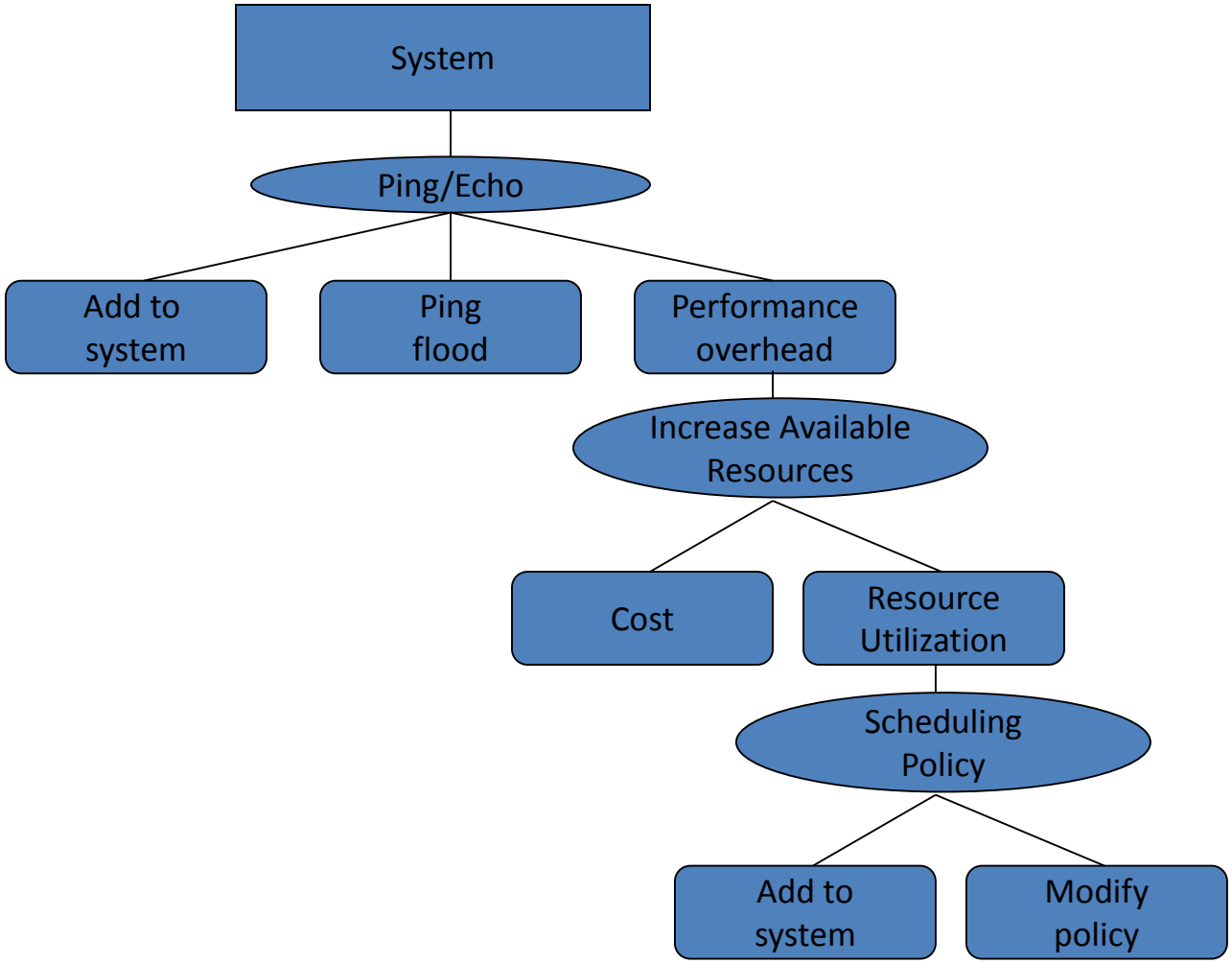
A tactic to address the efficient use of resources side-effect is “Scheduling Policy”.

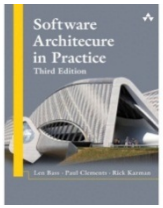
Common side effects of Scheduling Policy are:

- modifiability: how to add the scheduling policy to the existing architecture
- modifiability: how to change the scheduling policy in the future?



Tactics and Interactions - 7



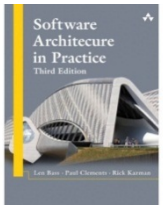


Tactics and Interactions - 8

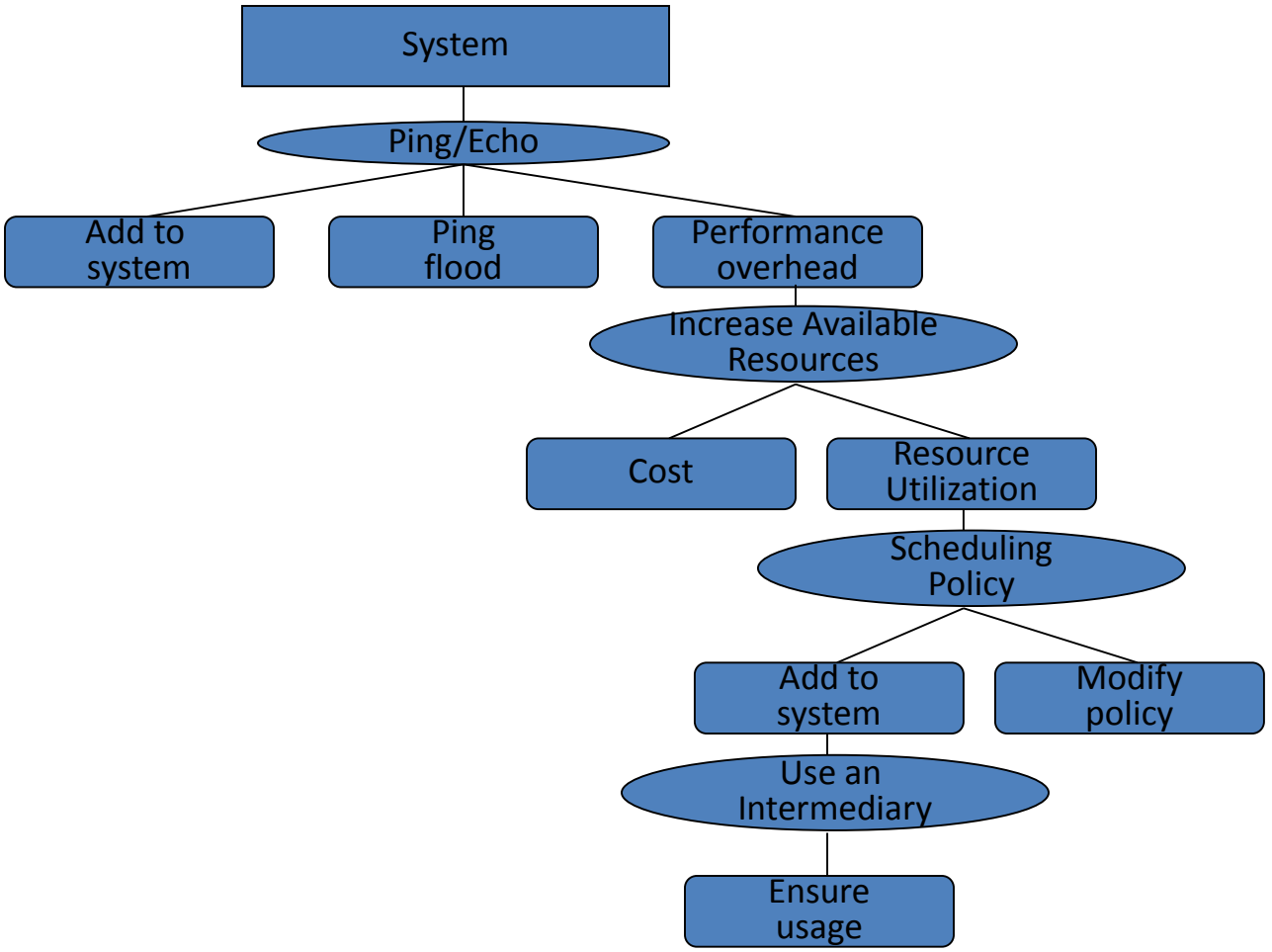
A tactic to address the addition of the scheduler to the system is “Use an Intermediary”.

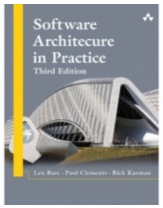
Common side effects of Use an Intermediary are:

- modifiability: how to ensure that all communication passes through the intermediary?



Tactics and Interactions - 9





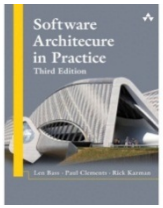
Tactics and Interactions – 10.

A tactic to address the concern that all communication passes through the intermediary is “Restrict Communication Paths”.

Common side effects of Restrict Communication Paths are:

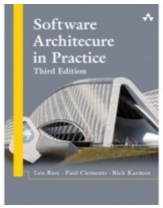
- performance: how to ensure that the performance overhead of the intermediary are not excessive?

Note: this design problem has now become recursive!



How Does This Process End?

- Each use of tactic introduces new concerns.
- Each new concern causes new tactics to be added.
- Are we in an infinite progression?
- No. Eventually the side-effects of each tactic become small enough to ignore.



Summary

- An architectural pattern
  - is a package of design decisions that is found repeatedly in practice,
  - has known properties that permit reuse, and
  - describes a *class* of architectures.
- Tactics are simpler than patterns
- Patterns are underspecified with respect to real systems so they have to be augmented with tactics.
  - Augmentation ends when requirements for a specific system are satisfied.