References

CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 18 Pipe Filters and Broker

Contents

Pipes & Filters ............................................................................................................. 2
The Broker Pattern ................................................................................................. 10
Dynamics .................................................................................................................. 12
Variants .................................................................................................................... 14
Known Uses ............................................................................................................. 16
Consequences .......................................................................................................... 17
Some RMI .................................................................................................................. 20
The Remote Interface .............................................................................................. 20
The Server Implementation ..................................................................................... 21
The Client Code ....................................................................................................... 24
Running The Example ............................................................................................ 25
Server Side ............................................................................................................... 25
Client Side .............................................................................................................. 27
Proxies ..................................................................................................................... 28

Reference

Pattern-Oriented Software Architecture, Buschmann et al.,

Copyright ©, All rights reserved. 2004 SDSU & Roger Whitney, 5500 Campanile Drive, San
Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.
Pipes & Filters

Unix Example

```
ls | grep -i b | wc -l
```

Context

Processing data streams

Problem

Building a system that processes or transforms a stream of data

Forces

- Small processing steps are easier to reuse than large components
- Non-adjacent processing steps do not share information
- System changes should be possible by exchanging or recombining processing steps, even by users
- Final results should be presented or stored in different ways
Solution

Divide task into multiple sequential processing steps or filter components

Output of one filter is the input of the next filter

Filters process data incrementally

- Filters does not wait to get all the data before processing

Data source – input to the system

Data sink – output of the system

Pipes - connect the data source, filters and data sink

Pipe implements the data flow between adjacent processes steps

Processing pipeline – sequence of filters and pipes

Pipeline can process batches of data
A Filter can be triggered by:

- Subsequent pipeline element pulls output from the filter
- Previous pipeline element pushes new data into filter
- Active Filter – pulls data from input and pushes data down the pipeline

If two active filters are adjacent the pipe between them synchronizes them
Some Implementation issues

Dividing the system into separate tasks

Data format passed between filters

This may require filters to convert from common format to a usable format

Implementing the pipes

Filter could directly push/pull data from another filter

Using a separate pipe mechanism
• More flexible
• Makes it easier to test filters
• Permits active filters
import java.util.ArrayList;
public class SharedQueue {
    ArrayList elements = new ArrayList();

    public synchronized void append( Object item )
    {
        elements.add( item);
        notify();
    }

    public synchronized Object get( )
    {
        try
        {
            while ( elements.isEmpty() )
                wait();
        }
        catch (InterruptedException threadIsDone )
        {
            return null;
        }
        return elements.remove( 0);
    }

    public int size()
    {
        return elements.size();
    }
}
Error handling

Difficult to do

What happens if 1/2 data is processed when one filter has a runtime exception?

How does one inform the other filters?

Can one restart the pipeline to process the next batch of data?
Example

d_eof := $@.

upperCaseFilter := 
[:input :output |
 | \nextCharacter |

[nextCharacter := input next.
nextCharacter ~= e_eof]
whileTrue: [output nextPut: nextCharacter asUppercase]]

noBeesFilter :=
[:input :output |
 | \nextCharacter |

[nextCharacter := input next.
nextCharacter ~= e_eof]
whileTrue:
ifTrue: [output nextPut: nextCharacter]]

display :=
[:input |
 | \nextCharacter |

[nextCharacter := input next.
nextCharacter ~= e_eof]
whileTrue: [Transcript nextPut: nextCharacter; flush]].
pipeA := SharedQueue new.
pipeB := SharedQueue new.

The Broker Pattern

Client-side Proxy
- packData()
- unpackData()
- sendRequest()
- return()

Broker
- updateRepository()
- registerService()
- acknowledgement()
- findServer()
- findClient()
- forwardRequest()
- forwardResponse()

Server-side Proxy
- packData()
- unpackData()
- callService()
- sendResponse()

Client
- callServer()
- startTask()
- useBrokerAPI

Bridge
- packData()
- unpackData()
- forwardMessage()
- transmitMessage()

Server
- initialize()
- registerService()
- runService()
- useBrokerAPI

calls

uses API

transfers message

transfers message

calls

uses API
A broker

- Handles the transmission of requests from clients to servers
- Handles the transmission of responses from servers to clients
- Must have some means to identify and locate server
- If server is hosted by different broker, forwards the request to other broker
- If server is inactive, active the server
- Provides APIs for registering servers and invoking server methods

Bridge

- Optional components used for hiding implementation details when two brokers interoperate
Dynamics

Registering Server

Server

start()
initialize()
registerService()
acknowledgement()
EnterMainLoop()

Broker

updateRegistry()
**Client Server Interaction**

Client-side Proxy

- `callServer()`
- `send Request()`
- `unpack Data()`
- `forward Request()`
- `return`
- `unpack Data()`

Broker

- `findServer()`
- `callService()`
- `forward Response()`
- `findClient()`

Server-side Proxy

- `unpackData()`
- `runService()`
- `packData()`

Server

- `packData()`

Client

- `result`

**Possible process boundary**
Variants

Direct Communication Broker System
• Broker gives the client a communication channel to the server
• Client and server interact directly
• Many CORBA implementation use this variant

Message Passing Broker System
• Clients and servers pass messages rather than services (methods)

Trader System
• Clients request a service, not a server
• Broker forwards the request to a server that provides the service

Adapter Broker System
• Hide the interface of the broker component to the servers using an additional layer
• The adapter layer is responsible for registering servers and interacting with servers
• For example if all server objects are on the same machine as application a special adapter could link the objects to the application directly
Callback Broker System

- Eliminate the difference between clients and servers
- When an event is registered with a broker, it calls the component that is registered to handle the event
Known Uses

CORBA
IBM's SOM/DSOM
Mircosoft OLE 2.x
RMI
Consequences

Benefits

Location Transparency
Clients (servers) do not care where servers (clients) are located

Changeability and extensibility of components
Changes to server implementations are transparent to clients if they don't change interfaces
Changes to internal broker implementation does not affect clients and servers
One can change communication mechanisms without changing client and server code

Portability of Broker System
Porting client & servers to a new system usually just requires recompiling the code
Benefits - Continued

Interoperability between different Broker System

Different broker systems may interoperate if they have a common protocol for the exchange of messages

DCOM and CORBA interoperate
DCOM and RMI interoperate
RMI and CORBA interoperate

Reusability

In building new clients you can reuse existing services
Liabilities

Restricted Efficiency

Lower fault tolerance compared to non-distributed software

Benefits and Liabilities

Testing and Debugging

A client application using tested services is easier to test than creating the software from scratch

Debugging a Broker system can be difficult
Some RMI

A First Program - Hello World

Modified from "Getting Started Using RMI"

The Remote Interface

```java
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```
The Server Implementation

// Required for Remote Implementation
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

// Used in method getUnixHostName
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HelloServer
        extends UnicastRemoteObject
        implements Hello
{


    public HelloServer() throws RemoteException
    {
    }

    public String sayHello() throws RemoteException
    {
        return "Hello World from " + getUnixHostName();
    }
}
protected String getUnixHostName()
{
    try
    {
        Process hostName;
        BufferedReader answer;

        hostName = Runtime.getRuntime().exec( "hostname" );
        answer = new BufferedReader( new InputStreamReader( hostName.getInputStream() ) );

        hostName.waitFor();
        return answer.readLine().trim();
    }
    catch (Exception noName)
    {
        return "Nameless";
    }
}
public static void main(String args[]) {
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());

    try {
        HelloServer serverObject = new HelloServer();

        Naming.rebind("//roswell.sdsu.edu/HelloServer", serverObject);

        System.out.println("HelloServer bound in registry");
    }
    catch (Exception error) {
        System.out.println("HelloServer err: ");
        error.printStackTrace();
    }
}
The Client Code

```java
import java.rmi.*;
import java.net.MalformedURLException;

public class HelloClient
{
    public static void main(String args[])
    {
        try {
            Hello remote = (Hello) Naming.lookup("//roswell.sdsu.edu/HelloServer");

            String message = remote.sayHello();
            System.out.println( message );
        }
        catch ( Exception error)
        {
            error.printStackTrace();
        }
    }
}
```

Note the multiple catches are to illustrate which exceptions are thrown
Running The Example
Server Side

Step 1. Compile the source code

Server side needs interface Hello and class HelloServer

javac Hello.java HelloServer.java

Step 2. Generate Stubs and Skeletons (to be explained later)

The rmi compiler generates the stubs and skeletons

rmic HelloServer

This produces the files:

HelloServer_Skel.class
HelloServer_Stub.class

The Stub is used by the client and server
The Skel is used by the server

The normal command is:

rmic fullClassname
Step 3. Insure that the RMI Registry is running

For the default port number

   rmiregistry &

For a specific port number

   rmiregistry portNumber &

On a UNIX machine the rmiregistry will run in the background and will continue to run after you log out

This means you manually kill the rmiregistry

Step 4. Register the server object with the rmiregistry by running HelloServer.main()

   java HelloServer &
Client Side

The client can be run on the same machine or a different machine than the server.

**Step 1.** Compile the source code

Client side needs interface Hello and class HelloClient

```
javac Hello.java HelloClient.java
```

**Step 2.** Make the HelloServer_Stub.class is available

Either copy the file from the server machine

or

Compile HelloServer.java on client machine and run rmic

**Step 3.** Run the client code

```
java HelloClient
```
**Proxies**

How do HelloClient and HelloServer communicate?

Client talks to a Stub that relays the request to the server over a network

Server responds via a skeleton that relays the response to the Client