Extension and Java Implementation of the Reactor−Acceptor−Connector Pattern Combination*

Alexandre Delarue
Department of Ocean Engineering
Florida Atlantic University, Dania Beach, FL 33004
adelarue@iname.com

Eduardo B. Fernandez
Department of Computer Science and Engineering
Florida Atlantic University, Boca Raton, FL 33431
ed@cse.fau.edu

Abstract

Development of efficient client−server applications involves good knowledge of multithreading and network communication, domains often closely related to the operating system. Therefore, the reusability of components could be limited by some system specific techniques or limitations used in the design. To enhance the reusability and ease of use, it is then very important to decouple communication establishment and initialization from service use in a client−server application. In this way, upgrades are simplified and one can improve, for instance, the message format without knowing the way services work. On the other hand a developer in charge of maintaining or creating a new kind of service does not need to know how the connections are concretely established. The composition of the Reactor pattern [2] and the Acceptor−Connector pattern [1] provides a powerful pattern to handle concurrent requests delivered to a server with the benefit of the separation of concerns between connection establishment and service use. We tried to enhance the portability of this solution by considering a Java implementation of this composition. But the Java language presents significant differences with C++, which lead us to modify the original patterns designed by D. C. Schmidt and to introduce a new pattern based on the Reactor pattern to solve the encountered problems.

Intent

The Reactor−Acceptor−Connector design pattern offers two main features inherited from the Reactor [2] and the Acceptor−Connector [1] patterns. First, it handles service requests that are delivered concurrently to an application by one or more clients. Then, it decouples connection establishment and service initialization in a distributed system from the processing performed once a service is initialized. The Reactor is in charge of dispatching the events received on registered handlers. The connection establishment and initialization is made by the Connector and Acceptor, the service is then performed by the initialized service handlers using the connection created by the Acceptor and the Connector.

Motivation

The department of Ocean Engineering of Florida Atlantic University needed a way to access and process information collected by AUVs (Autonomous Underwater Vehicles). The accessibility of these data is provided by a specific server and the data are processed using some GIS (Geographical Information System) software. The possibility to decouple the connection establishment and service initialization from the processing performed is really important here. It provides the department a certain freedom to use another GIS software (GRASS [14] is actually used) whenever it is needed. The modularity of this pattern allow the use of the same strategy to establish connections and initialize services. The only change is the way of processing data in the service handlers. The use of a Reactor to handle the client requests provides a good way to dispatch the events, avoiding a multithreaded approach (one thread per service) to concentrate on simplicity (a multithreaded server involves synchronization, context switching...) and adaptability. Indeed the Reactor design pattern offers simple ways to add, improve a service or change message formats without modifying the existing services; moreover, it gives the opportunity to use multithreading when needed.

* Copyright © 1999, Alexandre Delarue & Eduardo B. Fernandez. Permission is granted to copy for the PLoP 1999 conference. All other rights reserved.

Applicability

This pattern is more likely to be used in applications where the level of independence between the services and the connections must be high. This occurs when a software need to be adaptable to various network systems or to be easily upgraded by modifying the events model. The pattern also applies very well to systems where multithreading is inapplicable or should be limited (typically when one thread per service would lead to overuse of CPU resources due to excessive context switching). The use of the Java language to implement this pattern makes it also a great candidate for deployment of Intranet/Internet client–server applications via Java enabled web browsers.

Structure

Participants

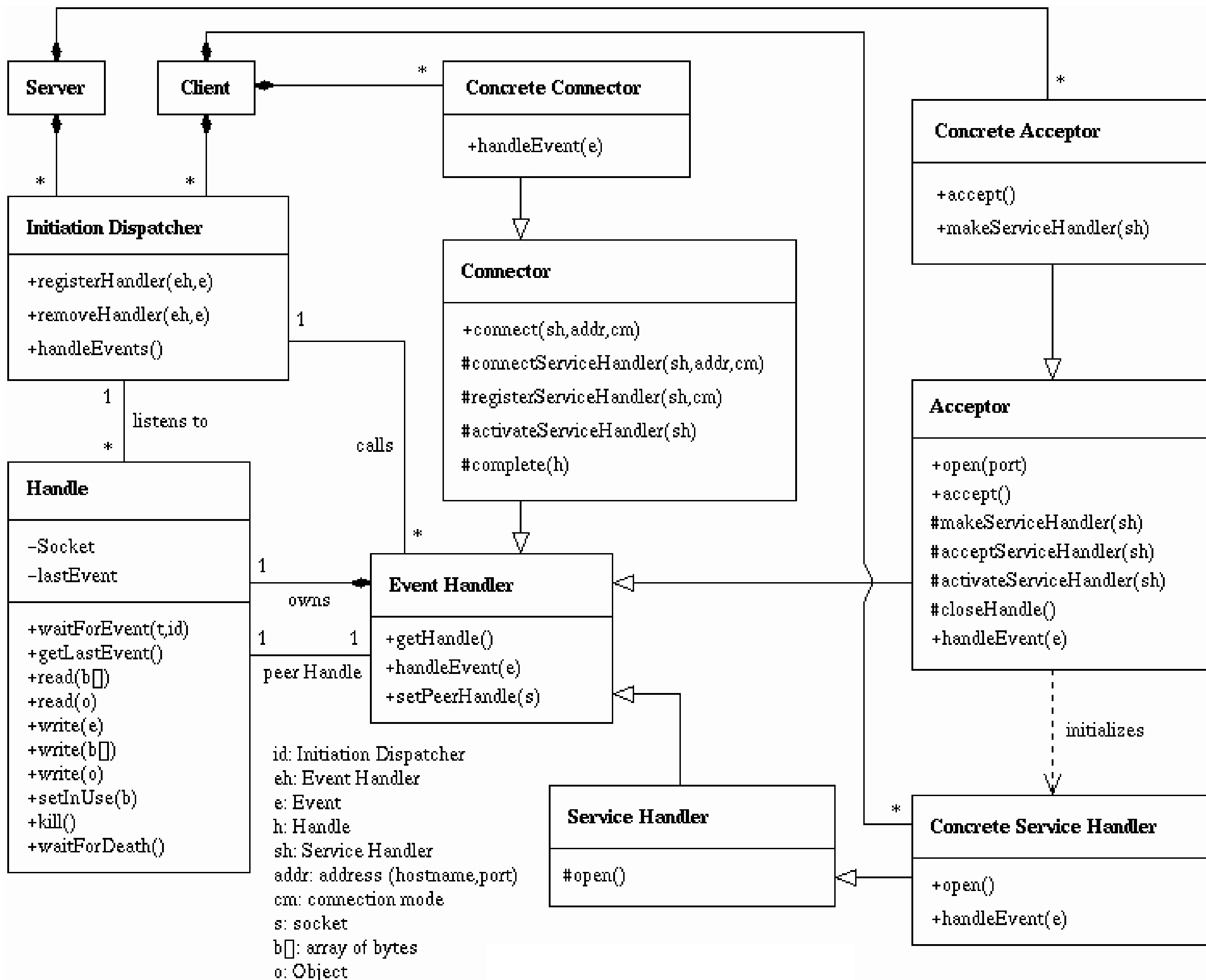

Fig.1 *– Participants of the Reactor–Acceptor–Connector pattern combination*

Compared to the original Reactor pattern [2] and Acceptor–Connector pattern [1], two major modifications (Fig.1) are noticeable. The Synchronous Event Demultiplexer entity is not present and the Handle entity encapsulates all the network specific access methods. These choices will be discussed in the Implementation section because they are mainly due to the choice of Java for the implementation.

Handle: Encapsulates the resources that are managed by the OS for communication. The Handle offers all the methods to access the communication resources so that the Initiation Dispatcher can wait for events to occur on them and the Service Handlers can use them to communicate with their peers. The aim of the Handle here is to concentrate network specific techniques and therefore to provide a high level network access.

Event Handler: Specifies an interface consisting of a hook method that abstractly represents the dispatching operation for service−specific events. This method must be implemented by application−specific services (Service Handlers).

Initiation Dispatcher: Is in charge of registering, removing and dispatching the events that occurs on the registered handles by calling back application−specific Event Handlers. Several Initiation Dispatchers can be created both on the server or on the client side to share the management of the registered Event Handlers.

Service Handler: Implements an application service, typically playing the client role, server role or both roles. It provides a hook method that is called by an Acceptor or Connector to activate the application service when the connection is established. The Service Handler uses the Handle component referencing to its peer for data−mode exchange with its connected peer Service Handler.

Acceptor: Is a factory that implements a passive strategy to establish a connection and initialize the associated Service Handler. It incorporates a passive mode endpoint transport factory (Handle factory) that creates the network endpoints needed by the Service Handlers.

Connector: Is a factory that implements an active strategy to establish a connection and initialize the associated Service Handler. It initiates the connection with a remote Acceptor and can use two connection modes to do so: a synchronous mode and an asynchronous mode.

Collaborations

Server side:

1/ Initialization phase

The server creates an Initiation Dispatcher that will handle the events associated with this server. It has then to create one Acceptor for each type of service it is offering to its clients. Each Acceptor registers itself to the Initiation Dispatcher so that it can handle an OPEN_SERVICE event coming from a client on the Acceptor Handle. The server can then ask the Initiation Dispatcher to handle all the registered events by listening to the associated Handles.

2/ Service initialization

The Initiation Dispatcher detects an OPEN_SERVICE event on the Handle of some Acceptor. It then calls the handle event method of this Acceptor for him to handle this particular event. The Acceptor creates a new service handler that will be in charge of processing the requests of the client. This Service Handler registers itself with the Initiation Dispatcher so that it could handle the following events: SERVICE_REQUEST, SEND_SERVICE_RESULT. Then it sends a SERVICE_READY event to its client peer Handle.

3/ Service use

The Initiation Dispatcher detects a SERVICE_REQUEST event on the Handle of one Service Handler. As a result of this, it calls its handle event method. The service then processes the request and sends a SERVICE_RESULT_READY to its client peer. When the Initiation Dispatcher detects a SEND_SERVICE_RESULT on the Handle of the Service Handler, it calls back the handle event method of this service. This Service Handler can then send the results to the client peer.

4/ Service closure

When the Initiation Dispatcher detects a CLOSE_SERVICE event on the Handle of a Service Handler, it calls its handle event method. The Service Handler has then to call the remove handler method of the Initiation Dispatcher to make it stop listening on its Handle for the events for which it had registered before. It can then close itself.

Client side (synchronous connection mode):

1/ Initialization phase

The client creates an Initiation Dispatcher that will handle the events associated with this client. Then it has to create one Connector and one Service Handler (that will be in charge of requesting the server peer service handler) for each type of service needed. For this, the Connector generates an OPEN_SERVICE event and sends it on the Handle of the server Acceptor in charge of the desired service. In return, the Connector gets a connection end point that corresponds to the Handle of the peer Service Handler. It can then open the client Service Handler. This can then register itself with the Initiation Dispatcher for the following event types: SERVICE_READY, SERVICE_RESULT_READY. The client can then call the handle events method of the Initiation Dispatcher.

2/ Service use

When the Initiation Dispatcher detects a SERVICE_READY event on the Handle of the client service handler, it calls the handle event method of this Service Handler. This one can then generate when needed a SERVICE_REQUEST event on its peer Service Handler's Handle. When the Initiation Dispatcher detects a SERVICE_RESULT_READY on the client service handler, it calls its Handle event method. The Service Handler then gets the results from its server peer.

3/ Service closure

If a client Service Handler wants to close the service, it generates a CLOSE_SERVICE event on its peer Service Handler's Handle. Then it has to remove itself from the handlers list of the Initiation Dispatcher by calling the remove handler method. It can then close itself.

Client side (asynchronous connection mode):

1/ Initialization phase

The client creates an Initiation Dispatcher that will handle the events associated with this client. Then it has to create one Connector and one Service Handler (that will be in charge of requesting the server peer Service Handler) for each type of service needed. For this, the Connector generates an OPEN_SERVICE event and sends it on the Handle of the server Acceptor in charge of the desired service. It then registers itself to the Initiation Dispatcher for the following event type: PEER_SERVICE_HANDLE_INITIALIZED. The client can then call the handle events method of the Initiation Dispatcher. This one will then notify the Connector when a PEER_SERVICE_HANDLE_INITIALIZED event will occur, so that the Service Handler can be activated by the Connector. This new Service Handler then registers itself to the Initiation Dispatcher for the following events: SERVICE_READY, SERVICE_RESULT_READY.

2/ Service use

When the Initiation Dispatcher detects a SERVICE_READY event on the Handle of the client Service Handler, it calls the handle event method of this Service Handler. This one can then generates a SERVICE_REQUEST event on its peer Service Handler's Handle. When the Initiation Dispatcher detects a SERVICE_RESULT_READY on the client Service Handler, it calls its handle event method. The Service Handler then gets the results from its server peer.

3/ Service closure

If a client Service Handler wants to end the service, it generates a CLOSE_SERVICE event on its peer Service Handler's Handle. Then it has to remove itself from the list of handlers of the Initiation Dispatcher by calling the remove handler method. It can then close itself.

Consequences

Benefits and drawbacks are mainly the combination of those of the Reactor [2] and the Acceptor–Connector [1] patterns. Here are the differences introduced with the use of Java for the Implementation.

Benefits

Improved configurability and extensibility: The modification of the communication system only requires to modify the Handles.

Improved portability: The service use is totally independent of the communication system between the client and the server. Portability problems will be concentrated on platform specific handle or network protocol, but the use of Java for the implementation tends to totally eliminate these problems as this pattern will work on every Java virtual machine with neither any modification nor need to recompile it for a new operating system.

Drawbacks

Restricted applicability: The use of this pattern with the Java implementation requires the presence of a Java Virtual Machine (JVM) on the operating system. Even if every operating system now tends to have at least one available JVM, some platforms do not have yet a reliable, optimized and up to date one, which could limit the applicability of this pattern. Moreover, a Java implementation involves higher memory and CPU usage than an equivalent C++ implementation. This could be limiting in situations where maximum performance is needed or when memory or CPU performance are limited.

Implementation: variations due to the use of Java

The Handle object

First, the lack of templates in Java has been compensated by encapsulating all the network oriented methods in the Handle object, so that the independence of the original patterns from the network aspects is preserved. Therefore, in this model, there is no direct network call in any entity but the Handle. This entity manages the creation, use and destruction of the system resources associated with the communication system, so that we concentrate communication changes in only one entity if needed. By doing this, we manage to preserve the flexibility of the Reactor pattern [2] and the Acceptor–Connector pattern [1] originally created by D. C. Schmidt with a C++ implementation.

This class encapsulates the network endpoint (a socket in this case) and contains all the methods to access this system resource. The creation of events is located in the Event Handlers (Service Handler, Acceptor, Connector). Typically, to generate an event on a peer Service Handler, a service uses its peer Handle, corresponding to the network endpoint of its peer, to send events via the write method. The peer Handle of a Service Handler is set during the initialization phase of the services by the Acceptor on the server side and by the Connector on the client side.

The *waitForEvent* method is in charge of detecting events on one Handle and is used by the Initiation Dispatcher. The parameters are the timeout value, to prevent from blocking situation on Handles, and a reference to the Initiation Dispatcher which is in charge of dispatching the incoming events occurring on the Handle.

The read and write methods provide simple ways for basic I/O access on each Handle.

To remove one Handle, one need to call the kill method so that no Initiation Dispatcher could listen anymore on the selected Handle. Then one has just to call the *waitForDeath* method. When it returns, the Handle is no more in use by any Initiation Dispatcher and can be safely cleared. This implementation lets a developer use the pattern without taking care of socket management.

The Synchronous Event Demultiplexer

The Synchronous Event Demultiplexer of the Reactor pattern [2] is not directly implementable in Java. There is no equivalent to the *select()* function used in C++ to specify the set of handles that will be handled concurrently by the operating system. For this reason, only two solutions seem easily applicable, either one uses one thread per Handle, or uses a Handle queue. The first solution should not be considered, as it quickly involves non negligible system resources consumption as the number of client increases. The second solution is a good starting point for its simplicity and is a reasonable solution for the specific GIS application which was not supposed to handle more than a few clients at the same time. Moreover the ability to use for instance several Initiation Dispatchers on the server at the same time guarantees a certain manageability of the overall performances in a multiple clients context.

The Initiation Dispatcher extends the Thread object of the Java programming language so that the *handleEvents* method creates a thread to listen to the Handles. This enables for instance the Client to handle its GUI after calling the non blocking *handleEvents* method of the Initiation Dispatcher. For this a simple method (which may not be optimal) is used. It consists of looking for incoming events on each registered Handle, one after another. One can specify the maximum time (timeout) to wait on each Handle, which can be used to tune the performance of the Initiation Dispatcher. The more the system waits, the less CPU resources it uses, but the more the average response time to an event is increased. This solution has been tested with a low timeout (10 to 50 ms) and gives good performance for a reasonable number of clients connected on a server. One could also use the possibility offered by the Reactor pattern [2] to use several Initiation Dispatchers when the number of services requested becomes too high. This provides a solution that consists of polling the Handles with a limited number of threads created according to need.

Sample Code and Usage

We describe now what is different in the implementation compared with the original Reactor [2] and Acceptor–Connector [1] patterns. Most of the components are easy to translate from C++ to Java and only the Handle and the Initiation Dispatcher's *handleEvents* loop are shown.

Handle

```
package RAC;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.net.SocketException;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;


/**
 * Defines a Handle.
 * @author Alexandre Delarue
 */
public class Handle
{
  private Socket       socket = null;
  private ServerSocket serverSocket = null;

  private Event        lastEvent = null;

  private boolean      isInUse = false;
  private boolean      isDying = false;


  /**
   * Creates a new Handle
   * @param addr The Address associated with this new Handle.
   */
  public Handle(Address addr)
  {
    try { this.socket = new Socket(InetAddress.getByName(addr.getHost()), addr.getPort()); }
    catch(IOException ioe) { System.out.println(ioe.getMessage()); }
  }


  /**
   * Creates a new Handle
   * @param port The port (on local machine) associated with this new Handle.
   */
  public Handle(int port)
  {
    try { this.serverSocket = new ServerSocket(port); }
    catch(IOException ioe) { System.out.println(ioe.getMessage()); }
  }


  /**
   * Creates a new Handle.
   * @param s The socket corresponding to this new Handle.
   */
  public Handle(Socket s) { this.socket = s; }


  /**
   * Wait for an event to occur on this Handle.
   * @param timeout The maximum time to wait before considering that no event occurs on this Handle.
   * @param id The Initiation Dispatcher that is handling the events.
   */
  public boolean waitForEvent(int timeout, InitiationDispatcher id)
  {
    if (socket != null)
    // Socket based Handle
    {
      try
      {
        socket.setSoTimeout(timeout);
        if (socket.getInputStream().available() > 0)
        {
          byte[] event = new byte[3];
          socket.getInputStream().read(event);
          if ((event[0] == Event.EVENT_BEGIN) && (event[2] == Event.EVENT_END))
          {
            lastEvent = new Event(event[1]);
            return true;
          }
          else
            return false;
        }
        else
          return false;
      }
      catch(SocketException se) { System.out.println(se.getMessage()); return false; }
      catch(IOException ioe) { System.out.println(ioe.getMessage()); return false; }
    }

    else
    // ServerSocket based Handle
    {
      try
      {
        serverSocket.setSoTimeout(timeout);
        Socket s = serverSocket.accept();
        lastEvent = new Event(Event.OPEN_SERVICE);
        ((Acceptor) id.eventInfoManager.getEventHandler(id.currentIndex)).setPeerHandle(new Handle(s));
        return true;
      }
      catch(SocketException se) { System.out.println(se.getMessage()); return false; }
      catch(IOException ioe) { System.out.println(ioe.getMessage()); return false; }
    }
  }
```

```java
  /**
   * Returns the last Event received on this Handle.
   * @return The Event.
   */
  public Event getLastEvent() { return this.lastEvent; }


  /**
   * To read some data from this Handle.
   * @param b The byte array used to store the read data.
   * @throws IOException
   */
  public void read(byte[] b) throws IOException
  {
    if (socket != null)
    {
      try { socket.getInputStream().read(b); }
      catch(IOException ioe) { throw ioe; }
    }
    else b = null;
  }


  /**
   * To read some data from this Handle.
   * @param o The object used to store the read data.
   * @throws IOException
   */
  public Object read() throws IOException
  {
    if (socket != null)
    {
      try
      {
        ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
        return ois.readObject();
      }
      catch(IOException ioe) { throw ioe; }
      catch(ClassNotFoundException cnfe) { throw new IOException(cnfe.getMessage()); }
    }
    else return null;
  }


  /**
   * To send an Event to this Handle.
   * @param et The Event to send to this Handle.
   * @throws IOException
   */
  public void write(Event e) throws IOException
  {
    if (socket != null)
    {
      try
      {
        byte[] event = new byte[3];
        event[0] = Event.EVENT_BEGIN;
        event[1] = e.getCode();
        event[2] = Event.EVENT_END;
        socket.getOutputStream().write(event);
      }
      catch(IOException ioe) { throw ioe; }
    }
  }


  /**
   * To write some data to this Handle.
   * @param b The byte array to send to this Handle.
   * @throws IOException
   */
  public void write(byte[] b) throws IOException
  {
    if (socket != null)
    {
      try { socket.getOutputStream().write(b); }
      catch(IOException ioe) { throw ioe; }
    }
  }


  /**
   * To write some data to this Handle.
   * @param o The object to send to this Handle.
   * @throws IOException
   */
  public void write(Object o) throws IOException
  {
    if (socket != null) {
      try
      {
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(o);
      }
      catch(IOException ioe) {throw ioe; }
    }
  }


  /**
   * Used to let the removeHandler method know that the InitiationDispatcher is using this Handle,
   * so that we can prevent it from being killed while used.
   * Call setInUse(false) to release the handle after use so that it can die if needed.
   * @param b True to set this Handle to isInUse, false to release this Handle.
   */
  public void setInUse(boolean b) { this.isInUse = b; }


  /**
   * Begin the process of removing the handle. No Initiation Dispatcher will be allowed to listen anymore on this handle.
   */
  public void kill() { this.isDying = true; }


  /**
   * Wait for the InintiationDispatcher to end listening on the Handle if the kill method was called while the Handle was in use.
   */
  public void waitForDeath() { while (this.isInUse); }
}
```

Initiation Dispatcher

This loop runs as a thread and is in charge of detecting the events occurring on the registered Handles. The *eventInfoManager* object is mainly managing the list of Events, Handles and Event Handlers; it can be considered as a part of the Initiation Dispatcher. Notice the use of the `h.setInUse(true)` and `h.setInUse(false)` technique to indicate when a Handle is being used by one Initiation Dispatcher. It prevents it from being removed while in use.

```
/**
 * The thread run() method that performs the handleEvents loop
 */
public void run()
{
  Handle h = null;
  Event  e = null;

  // loop that handles events
  while (IamAlive)
  {
    // what is the index of the next Handle to pool ?
    currentIndex = eventInfoManager.nextIndex();
    if (currentIndex > -1)
    {
      // get the Handle to listen on
      h = eventInfoManager.getHandle(currentIndex);
      // reserve the right to use the handle
      h.setInUse(true);
      // wait for an event to occur on it
      if (h.waitForEvent(timeout, this))
      {
        // an event has been detected
        e = h.getLastEvent();
        // call the corresponding Service Handler's handleEvent method
        eventInfoManager.getEventHandler(currentIndex).handleEvent(e);
      }
      // free the right of use
      h.setInUse(false);
      // makes the thread wait for a while before querrying another Handle
      // to limit CPU resource use
      try { sleep(pollingDelay); }
      catch(InterruptedException ie) { System.out.println(ie.getMessage()); }
    }
    else
    // there is no Handle available
    {
      // makes the thread wait for a while if there is no registered Handle
      // to limit CPU resource use
      try { sleep(noHandleAvailableDelay); }
      catch(InterruptedException ie) { System.out.println(ie.getMessage()); }
    }
  }
}
```

Discussion

Unsatisfying queuing method

The use of the handle queuing method to replace the Synchronous Event Demultiplexer becomes rapidly limited in terms of performance. Indeed as the number of Handles increases, the time needed by the Initiation Dispatcher to check two times the same Handle increases, so that the average response time of the server is bigger. To try to keep a good response time, one could use the ability of the Reactor pattern to use several Initiation Dispatchers. But this is not always very convenient as a proper estimation of their number is needed and as the Handles have to be correctly dispatched.

Modification and creation of a new pattern

Because of this, the Reactor pattern has been modified to maintain a certain level of performance with regards to the number of incoming events. This lead to a new pattern, the "Controlled Reactor", which offers two improvements of the Java implementation of the Reactor (Fig.2). It is based on three main components: the Handle Info Manager which takes care of the (un)registration of all the Handles, the Event Dispatcher(s) which is (are) polling the registered Handle to detect incoming events, and the Performance Manager which is checking the overall performance of the system and can decide to create or destroy one Event Dispatcher.

Main improvements

First, it gives the opportunity to use multithreading in a very transparent way for the developer. It lets you use several threads (the Event Dispatchers) to listen to the Handles without taking care of the way they are created and managed. In the Reactor pattern, you use multiple threads to manage the Handles by explicitly creating several Initiation Dispatchers and by dispatching the Handles to them. Here, the Controlled Reactor pattern provides an automatic mechanism to create/destroy threads to manage the handles in function of some specified performance constraints (for instance a minimum response time).

Then, this pattern solves the problem of a blocking *handleEvent* method that could occur on an Event Handler and locked all the mechanisms, without using one thread per Event Handler. Indeed, the Performance Manager will automatically detect blocking

situations as it will increase the response time. Indeed, if one Event Dispatcher is locked on the handleEvent method of one Event Dispatcher, the time between two calls of the *waitForEvent* method on one Handle will increase. The Performance Manager could then consider that one Event Dispatcher is frozen and that it needs to create a new one. Moreover if the blocking situation is in fact not a blocking situation but a service that needed a long time to process, the former blocked Event Dispatcher will be released once the handleEvent has completed. The Performance Manager could then decide to kill one of the Event Dispatchers if the last created one is no more necessary.

New Participants

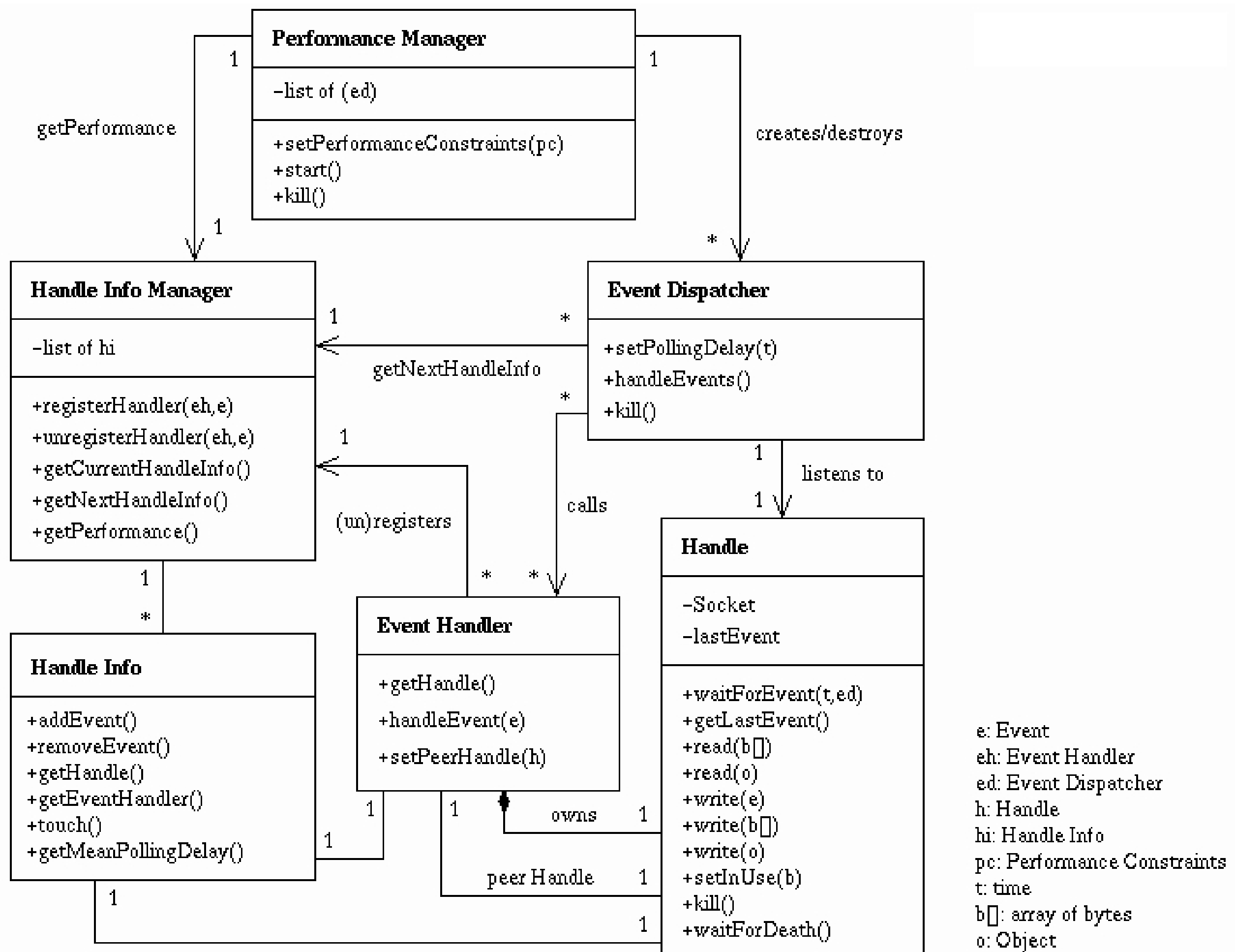

***Fig.2** – Participants of the Controlled Reactor pattern*

Handle Info: Maintains the references to one Event Handler, its associated Handle and the registered Event Types. The Handle Info entity is also in charge of determining the average time needed to poll this Handle, so that the Handle Info Manager can estimate the average performance of the system.

Handle Info Manager: Manages the list of Handle Info entities. The Event Handlers have to register or unregister with this participant. The Handle Info Manager is in charge of providing a reference to the next Handle (via the Handle Info) that must be listened to when an Event Dispatcher needs a new Handle in its *handleEvents* loop. So, the Handle Info Manager dispatches the Handle Info among the Event Dispatchers so that all the Handles are monitored as often as possible and only by one Event Dispatcher at the same time.

Event Dispatcher: Its aim is to detect the incoming events occurring on the Handles. For this, it continuously asks the Handle Info Manager for the next Handle to listen on, tries to detect an event on it and dispatches it to the right Event Handler when it occurs. Several Event Dispatchers can work at the same time (each one is using one thread) on the same set of Handles (managed by the Handle Info Manager).

Performance Manager: Lets the user of the pattern specify some performance constraints on the system. Performance is mainly expressed by the definition of the maximum response time; that is, how much time will the system need to call the Event Handler's handleEvent method associated with one incoming Event. This basically corresponds to the delay between two *waitForEvent* calls on the same Handle and can be reduced by increasing the number of Event Dispatchers listening on the set of registered Handles. We could also specify some constraints on the system; for instance, the maximum number of Event Dispatchers allowed to work at the same time. The Performance Manager is then in charge of monitoring the activity of the Handle Info Manager to evaluate the response time of the system. Specifically, it needs to determine when it must create or destroy an Event Dispatcher according to the specified constraints.

Known Uses

The GIS (Geographical Information System) client–server application [13] of Florida Atlantic University (department of Ocean Engineering) has been designed using the Java implementation of the Reactor–Acceptor–Connector pattern combination (without the Controlled Reactor). This system is still under development and will be mainly based on a GIS engine called GRASS [14] which represents the core of the different services provided by this server. But the convenient management of services provided by this pattern makes this application very simple to extend by linking it with other specific software. The work is now centered on the integration of the Controlled Reactor. The need for such a pattern became higher as we performed some basic tests. Indeed, GIS processing often involves long computational time which highly reduces the efficiency of the server by locking for a while the *handleEvents* method of the Initiation Dispatcher. The Controlled Reactor solves this problem and provides management of the overall performance.

Obviously, the original patterns have been used in many systems such as those mentioned by D. C. Schmidt in [1] and [2], InterViews [6], UNIX network super servers (such as inetd [7] or listen [8]), CORBA ORBs [9], WWW Browsers, Ericsson EOS Call Center Management System [10], Project Spectrum [11],and the ACE Framework [12].

Related Patterns

The presented pattern is a combination and extension of existing patterns, therefore it is related to the same patterns as those mentioned in [1] and [2]: that is, mainly the Proactor [5] pattern and the Active Object [4] pattern.

References

[1] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services", in *Pattern Languages of Program Design* (R. Martin, F. Bushmann and D. Riehle, eds.). Reading, MA: Addison–Wesley, 1997.

[2] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), . 529–545. Reading, MA: Addison–Wesley, 1995.

[3] D. C. Schmidt, "A Family of Design Patterns for Application–Level Gateways", *Theory and Practice of Object Systems*, Vol. 2, No. 1. Wiley & Sons, December 1996

[4] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming" in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison–Wesley, 1996

[5] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers", in *The 4th Pattern Languages of Programming Conference* (Washington University technical report #WUCS–97–34), September 1997

[6] M. A. Linton and P. R. Calder, The Design and Implementation of Interviews, in *Proceedings of the USENIX C++ Wokshop*, November 1987.

[7] W. R. Stevens, *UNIX Network Programming, First Edition.* Englewood Cliffs, NJ: Prentice Hall, 1990.

[8] S. Rago, *UNIX System V Network Programming.* Reading, MA: Addison–Wesley, 1993.

[9] Object Management Group, *The Common Object Request Broker: Architecture and Specifications,* 2.0 ed., July 1995.

[10] D. C. Schmidt and T. Suba, "An Object−Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems", *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems),* vol.2, pp. 280−293, December 1994.

[11] G. Blaine, M. Boyd and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System", *HIMSS, Health Care Communications,* pp. 71−81, 1994.

[12] D. C. Schmidt, "ACE: an Object−Oriented Framework for Developing Distributed Applications", in *Proceedings of the 6th USENIX C++ Technical Conference,* (Cambridge, Massachusetts), USENIX Association, April 1994.

[13] A. Delarue, S. Smith and E. An, "AUV Data Processing and Visualization using GIS and Internet Techniques", in *Proceedings of the Oceans '99 MTS/IEEE Conference*, http://adelarue.tsx.org/

[14] Geographic Resources Analysis Support System (GRASS), Baylor University, Texas, http://www.baylor.edu/~grass/