A New Java Multi-threading Communication Mechanism

YANG LI, RUSSELL J. DEATON

The University of Memphis, Department of Electrical Engineering,Memphis, TN38111

yangli@memphis.edu

rjdeaton@memphis.edu

Abstract This article introduces a framework acts as Mediator and Façade[1] providing communication mechanism between threads acting as data producer or/and data consumer. Using this framework, users can concentrate on data processing design within threads and the framework will establish the connections. Data is transferred as object(s). The communication connection is simple, loose coupled and even can be dynamically binded at run-time. Any user has just basic Java knowledge can use this framework to take advantage of Java's integrated support for multi-threads programming. The communication mechanism for data communication through networks and an industrial application is also discussed. Three examples are provided.

1 Background

Java provides quite simple and tightly integrated support for thread, however this support is very primitive in Java core API. There are only a few basic facilities designed to support Java concurrent programming. As a result, writing concurrent programs in Java can be difficult and confusion. Designing data communication between threads needs lot efforts. To make things even worse, the data flow is intrinsic difficult to follow at run-time, debugging multi-threads usually becomes a nightmare. Those extra efforts shift programmer's focus away from the more important issues: data processing within threads.

In real world, a large portion of threads acts as data producer or/and data consumer. To establish connection between threads in those situations, there are three common approaches:

1. Using java.io Package, such as ObjectInputStream class, PipedInputStream class etc. This approach builds every thing from scratch. You have to pay a lot attention to many details. Sometimes you have to deal with data transform and recovery to/from streams. Those efforts have to be applied to every single thread. Any future changes are very difficult and trouble-prone.
2. Using Observer (Event/ActionListener) pattern. This approach is much easier and integrated. It is the basic and widely used Java data communication mechanism between threads. However this approach is still quite primitive, lot connection details are exposed to users. Furthermore, there is a big disadvantage: two threads are tightly coupled. Synchronization is needed.
3. Using InfoBus by Lotus. This is a good solution for enterprise application. However it mainly serves Java Beans and is an overkill for many small applications

All above approaches require user having a good knowledge about Java concurrent programming and putting lot energy in debugging connections. The learning curve is very high. In the first two approaches, connections are hard-coded and any change is difficult to make.

2 First Example Using New Framework

Here a framework that can greatly ease the pain designing and debugging multi-threads data communication under data producer/consumer model is presented. Firstly, Let us looks how this framework works by going through an example.

Suppose we are going to design a simulation of a simple temperature control system as Fig.1. There are two sensors that measure temperature every 12 seconds and send their measurement to a controller. The controller compares the two temperature measurements most recently received every 4 seconds and sends increase or decrease control signal to a heater accordingly. The heater increases or decreases heater power every 2 seconds depending on the control signal most recently received.

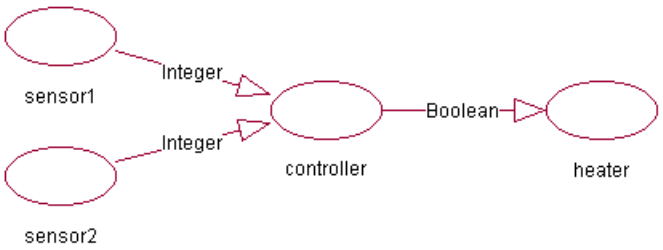

Fig. 1 Control System Example

The programs show below:

```
Sensor.java:
import jpal.*;
public class Sensor extends ThreadMonitor {
    // data channel used to send to a controller
    ObjectOutputChannel out1
        = new ObjectOutputChannel("Integer", "controller");
    public Sensor(String name) {
        super(name);
        addChannel(out1);
        setSleepTime(12*1000); // this thread generates a new
                               // temperature then sleeps for every 12s
    }

    public void process() {
        // generate a random number as temperature
        Integer temperature = new Integer((int)(10*Math.random()));
        out1.writeObject(temperature);
    }
}

Controller.java:
import jpal.*;
public class Controller extends ThreadMonitor {
    // channel receives temperature from sensor1
    ObjectInputChannel in1
        = new ObjectInputChannel("Integer", "sensor1");
    // channel receives temperature from sensor2
    ObjectInputChannel in2
        = new ObjectInputChannel("Integer", "sensor2");
    // channel sends control signal to a heater
    ObjectOutputChannel out1
        = new ObjectOutputChannel("Boolean", "heater");
    Integer sensor1 = new Integer(0);
    Integer sensor2 = new Integer(0);
    Boolean output = new Boolean(false);

    public Controller(String name) {
        super(name);
        addChannel(in1);
        addChannel(in2);
        addChannel(out1);
        setSleepTime(4*1000);
    }

    public void process() {
        if (hasInterrupted()) { // this thread is interrupted if
                                // there are input data
            if (in1.available()>0) {
                sensor1 = (Integer)in1.readObject();
                printInfo(" sensor1 T=" + sensor1.toString());
            }
```

```
            if (in2.available()>0) {
                sensor2 = (Integer)in2.readObject();
                printInfo(" sensor2 T=" + sensor2.toString());
            }
        } else { // if it is a sleep(sleepTime) time out
            if ((sensor1.intValue()-sensor2.intValue()) >= 0) {
                output = new Boolean(true);
            } else {
                output = new Boolean(false);
            }
        }
        out1.writeObject(output);
    }
}

Heater.java:
import jpal.*;
public class Heater extends ThreadMonitor {
    // channel receives control signal from controller
    ObjectInputChannel in1
        = new ObjectInputChannel("Boolean", "controller");
    int output = 0;
    Boolean control = new Boolean(true);

    public Heater(String name) {
        super(name);
        addChannel(in1);
        setSleepTime(2000);
    }

    public void process() {
        if (in1.available()>0) {
            control = (Boolean)in1.readObject();
            printInfo(" received control signal:"
                      + control.booleanValue());
        }
        if (control.booleanValue()) output++;
        else output--;
        printInfo(" heater: " + output);
    }
}

Register.java:
import jpal.*;
public class Register extends ThreadRegister {
    public Register() {
        super();
        ThreadMonitor thread1 = new Sensor("sensor1");
        ThreadMonitor thread2 = new Sensor("sensor2");
        ThreadMonitor thread3 = new Controller("controller");
        ThreadMonitor thread4 = new Heater("heater");
        register(thread1);
        register(thread2);
        register(thread3);
        register(thread4);
    }

    public static void main(String args[]) {
        Register reg = new Register();
        reg.init(args);
        reg.start();
    }
}
```

The bold lines are code lines that users need to establish the connection between threads. Basically, every thread subclasses a ThreadMonitor class that itself is a subclass of java.lang.Thread. For establishing data communication connection, users need to do following:
1. Declaring an ObjectInputChannel or/and ObjectOutputChannel. The first parameter of constructor is the object type you are going to transfer. The second parameter is the source or

destination thread name.
2. Using addChannel method to add input or output channels.
3. Using readObject or writeObject to transfer data.
4. Registering all threads by using register method in a subclass (Here we name it Register.java) of a ThreadRegister class.
5. Running the Register.class by typing "java Register". The ThreadRegister will take care all the connections for users and start all threads after that.

3 Structure of the Framework

There are two level in designing this framework. This section introduces the lower level: the data communication structure under same JVM. The higher level of the framework deals with the data communication issues among different hosts through networks (see section 4).

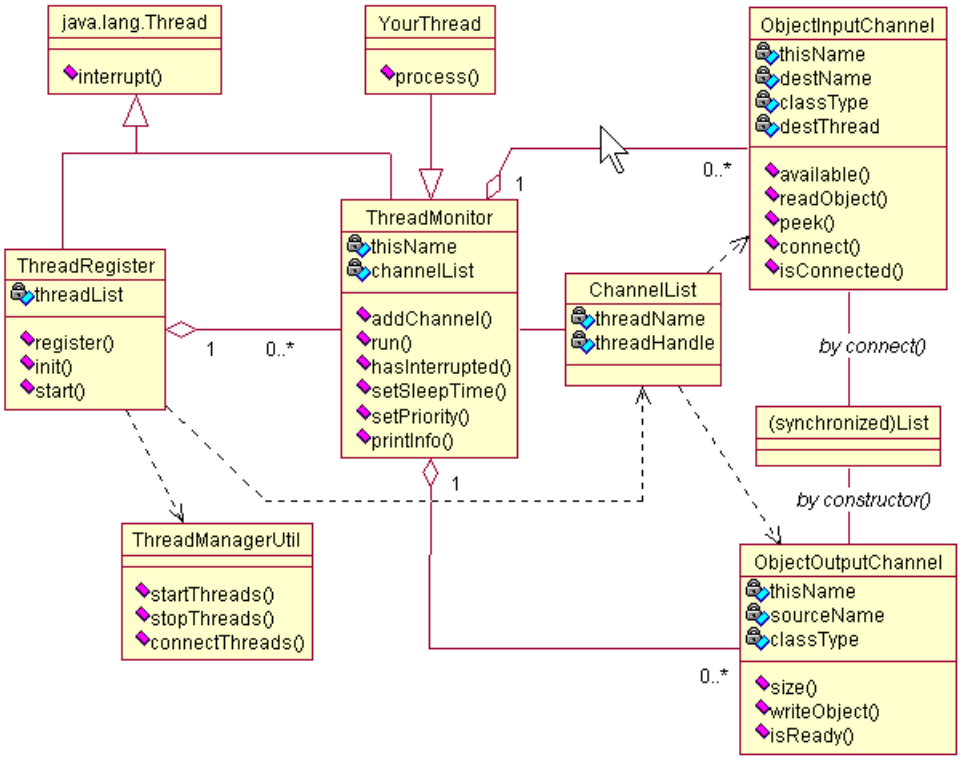

Fig. 2 Framework Structure in UML[2]

3.1 Data Transfer Channels

An ObjectOutputChannel creates a java.util.List that is shared by an ObjectInputChannel. The List serves as a data buffer between data producer and data consumer. When the data consumer is busy, data producer saves objects (the references) to be sent in the List buffer until the consumer can process them. User can use size() method of ObjectOutputChannel to control the length of the buffer list preventing it from growing too long. The data producer notifies destination thread by invoking data consumer's interrupt () method. If consumer thread is idle, it will be waken up immediately by an InterruptException. Otherwise, The consumer can check late by using isInterrupted () method. The producer transfers the object reference, not object itself to consumer, so complexity and size of the object to be sent do not matter. After producer added object to the List buffer, we can de-reference the object in producer, which make the object already sent not available in producer thread. By doing so we can reduce the coupling points between two threads to only two: The producer could invoke consumer's interrupt () method and both producer and consumer could visit the List buffer at same time (the List is generated by Collections.synchronizedList (ArrayList list), so this would not cause trouble). This minimizes multi-threading conflicts and simplifies program analyses.

3.2 Channels Connection

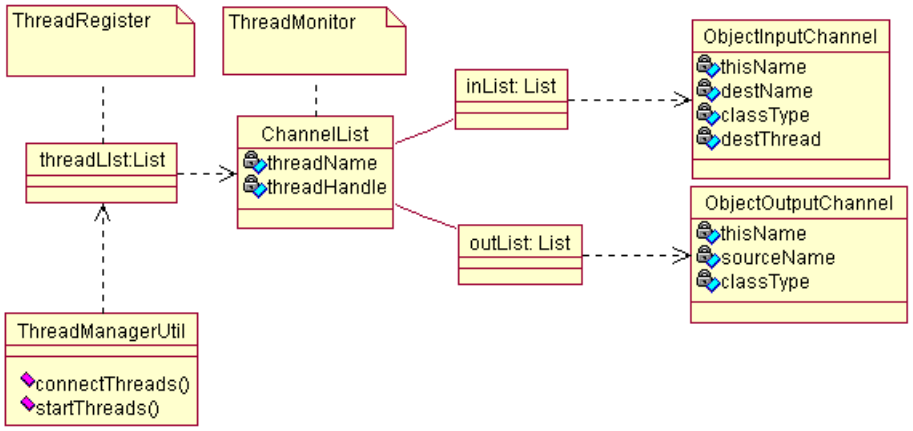

Fig. 3 Connection Information Structure

When your thread subclass ThreadMonitor, it inherits a ChannelList class that consists two java.util.List: one is a list of all ObjectInputChannels declared in the thread, another is a list of ObjectOutputChannels. All your threads is created in a ThreadRegister class, this ThreadRegister keeps a java.util.List consisting instances of ChannelList that are sent in by every your threads. Because all information about source threads and destination threads is contained in these instances of ChannelList, the ThreadRegister can make all connections between threads. The ThreadRegister class works as a "Mediator"(Form GoF, the definition of Mediator is "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently"). As a result, these connections can dynamically binded at run-time. Furthermore, the ThreadMonitor class also exposes this connection information through index property, so connections can be made by using a visual tool such as IBM's VisualAge for Java.

3.2 Fitting this Framework into your Application

Because All your threads need to subclass ThreadMonitor class, one question arises: how this single framework can fit different applications.
The following is the run () method of ThreadMonitor class:

```
public void run() {
    runLoop: while (connected) {
        try {
            if (stopMonitor) { break runLoop; } // stop thread by
                                                // return from
                                                // run()
            process(); // goes to your special message processing
            hasInterrupted = false;
            Thread.sleep(sleepTime);
        } catch(InterruptedException e) {
            hasInterrupted = true; // this flag tells if runLoop
                                   // restart because an
                                   // InterruptException is thrown
                                   // or just a sleep() timeout
        }
    }
}
```

If your thread subclasses ThreadMonitor class, you can override process () method. Code above shows that the process () method is executed under two conditions: when the thread receives an interrupt or Thread.sleep(sleepTime) timeout.
If your thread does some tasks like receiving a message, processing it and waiting for next message. This structure fits your need very well. You only need put all your data processing tasks

in a process () method in your thread and do some simple connection registration. Then everything is done. You can have most your energy spent on your special data processing programming.
If your thread is computational expensive or has complex structure, you can generate a new thread or new object to do your tasks. At the same time, let the framework works at background handling the data communication. The second example shows how to do this.

4 Communication between Threads at Different JVMs

The second level of the framework deals with the data communication issues among different hosts through networks. At the second level, A enhanced ThreadRegister called ThreadManager acts as a Facade. This Facade provides a single and simplified interface of the outside networks to those threads running at local JVM.

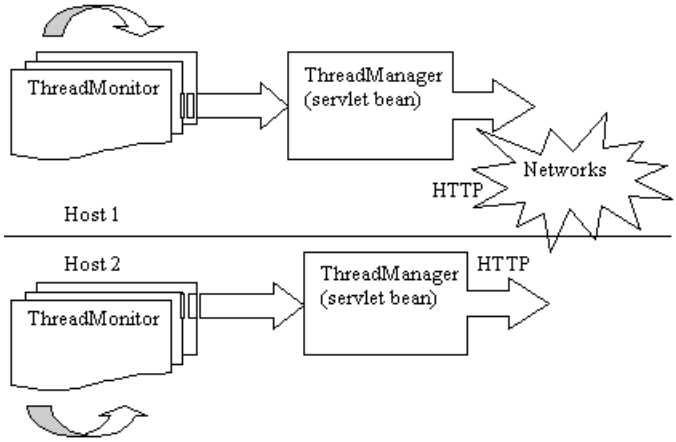

Fig. 4 Communication between JVMs

Two ThreadManagers at different hosts talk with each other through HTTP protocol. The ThreadManager runs as a servlet. Because HTTP and Java Servlets are supported by almost any system, It should be the best choice for this kind of loose-coupled data producer/data consumer communication. It also works if you would like to exchange data through a firewall. Although other mechanisms (like Java RMI etc.) can do the same job, the major concern here is keeping things simple and avoiding complex installation. If there is a tight connection needed here, we can also encapsulate a RMI connection(by HTTP)in the ThreadManager facade. The goal of the framework is shielding all complexity and details of network communication from user. For example, the ObjectInputChannel/ObjectOutputChannel constructor signature is same. The only difference the user should make is adding a destination host URL such as:

ObjectInputChannel in = new ObjcetInputChannel("String", "hostURL/threadName");

This will establish a connection channel between your threads and a ThreadManager. The object is transferred first to ThreadManager, then serialized and sent to destination host through HTTP protocol communication. From a user's view, There is almost no difference whether the destination thread is under the same JVM or running at another host.
By providing a design time BeanInfo class for ThreadMonitor class. This will let user make their design and connections under a visual builder environment.

5 An Industral Application Example

There is a data communication system in a company. The characteristic of the system is its heterogeneous mix of hardware and software, and network protocols. There are hundreds of independent agents communicate with a couple of hosts. Between those agents and hosts, there are some middleware servers providing data format convention, other data processing, data traffic controlling and routing. Agents exchange data with middleware servers through a public network. The middleware servers exchange data with hosts through a LAN.
Here is our Java solution with the framework:

We use two level modulation: service level and function level. The middleware server provides data processing and routing service. This is a typical flow pattern[3] problem in concurrent programming. Every thread(subclass of ThreadMonitor) is a function module as a Java bean. Each thread bean provides one or more data processing functions. Many function modules compose a servlet bean(subclass of ThreadManager). A couple of servlet beans works under a Java web server providing all the services.
The new framework greatly eases the maintenance efforts. You can add a new service by adding a new servlet bean. Or upgrading one of your services by changing function thread module in the servlet. As long as the framework rules are followed, No new efforts needed to establish and debug the connection again. Furthermore using a visual builder tool can do all these connections.

6 The Second Example

This example demonstrates how this framework works with swing components, and how easily user can transfer large size object like Image between threads.

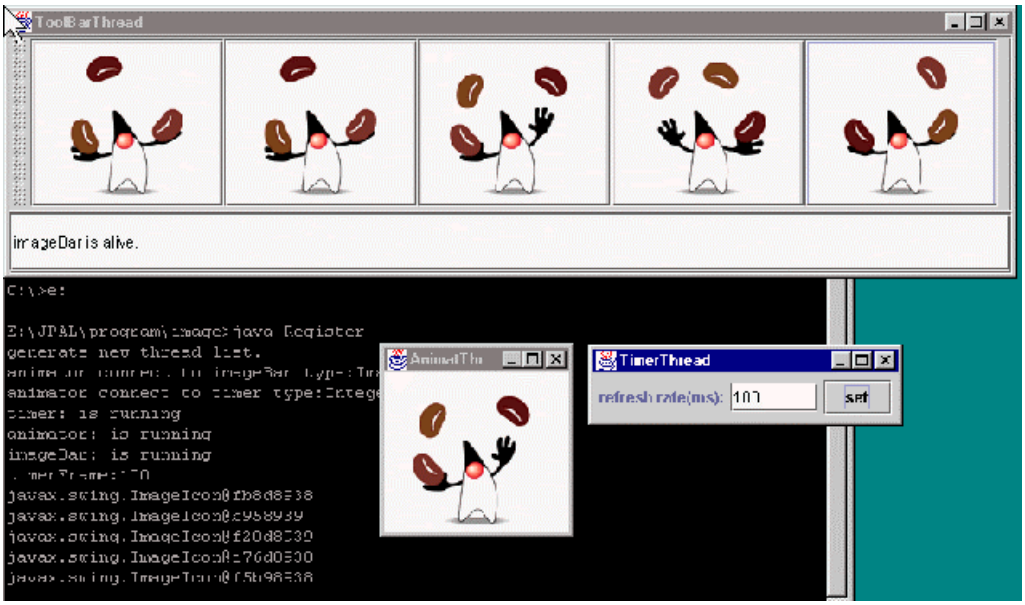

Fig. 5 Example2 is running

The structure of the second shows in Fig. 6. In this example, a model-view structure is used. Because we can not have a class extends both ThreadMonitor and JFrame, we let this framework keeps the Images and handles data communication at background. At the same time, we delegate Graphic tasks to GUI swing thread.
The ImageBar keeps 5 juggler images in a bar. Every time you click an image in the image bar, that image is sent to Animator. The Animator runs the animation loop. The Timer determines the refresh rate. The Animator runs repaint () method every time it receives an interrupt from Timer. This example also shows how you can subclass ThreadMonitor to do some very useful special tasks such as Timer. This is not an easy work when you use other mechanisms[4].

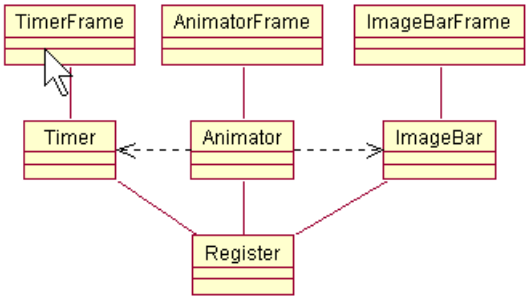

Fig. 6 Framework Works in the Background

```
Animator.java:
...
public class Animator extends ThreadMonitor {
    // animator receives image in ImageIcon format from imageBar
    ObjectInputChannel in1
        = new ObjectInputChannel("ImageIcon", "imageBar");
    // animator receives interrupt from timer
    ObjectInputChannel in2
        = new ObjectInputChannel("Integer", "timer");
    AnimatorFrame aAnimatorFrame; // new object for GUI tasks
    ...
    public void process() {
        if(hasInterrupted()) {
            if (in1.available()>0) {
                //reads image from image bar
                ImageIcon imageIcon = (ImageIcon)in1.readObject();
                ...
            }
            while (in2.available()>0) {
                // an interrupt from timer received
                Integer rate = (Integer)in2.readObject();
            }
        }
        // received a interrupt from timer, do refreshing.
        aAnimatorFrame.refresh();
    }
}

Timer.java:
public class Timer extends ThreadMonitor {
    // sends interrupt to animator, works as a timer
    ObjectOutputChannel out1
        = new ObjectOutputChannel("Integer", "animator");
    ...
    public Timer(String name) {
        super(name);
        addChannel(out1);
        ...
    }

    // process() is executed when super.sleep(sleepTime) time out
    public void process() {
    ...
        out1.writeObject(rate);
    }
}
```

7 The Third Example

Recently we are involved in a project in which we simulate network communication. We need to generate about 100 agents. Those agents randomly send data through a UDP port to a server. This is a good situation to use multi-threads programming. We found it is very easy work if we use the new framework. The following example is a simplified problem. We have 100 independent data source, each one sends a serial number to a port thread. The data output rate of data source is randomly chosen when the data source is initiated. A port thread receives data from the 100 data sources. Instead of sending the received data to a server, this port thread prints out the data to standard output. The following are classes we need:

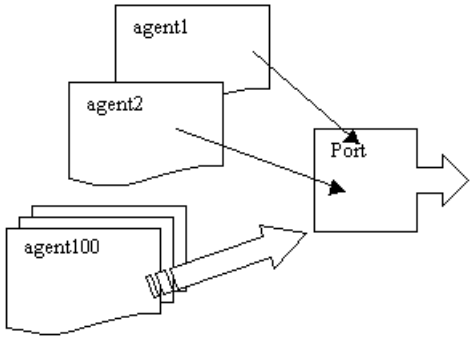

Fig. 7 Simplified Simulation Structure

```
Source.java:
import jpal.*;
public class Source extends ThreadMonitor {
    ObjectOutputChannel out1
        = new ObjectOutputChannel("Integer", "port");
    int sleepTime = 1000;
    int serial = 0;
    public Source(String name) {
        super(name);
        addChannel(out1);
        // generate a random data output rate: 1-100s
        sleepTime = (int)(100*1000*Math.random()) + 1000;
        setSleepTime(sleepTime);
    }
    public void process() {
        Integer data = new Integer(serial);
        // sends serial number to port
        out1.writeObject(data);
        serial++;
    }
}

Port.java:
import jpal.*;
public class Port extends ThreadMonitor {
    // this port receives data from 100 input channel
    int sourceNum = 100;
    ObjectInputChannel[] in = new ObjectInputChannel[sourceNum];
    public Port(String name) {
        super(name);
        // this thread receives data from 100 source agents
        for (int i=0; i<sourceNum; i++) {
            // sourceName: source thread name
            String sourceName = "source" + String.valueOf(i);
            in[i] = new ObjectInputChannel("Integer", sourceName);
            addChannel(in[i]);
        }
        setSleepTime(3600*1000);
    }
```

```
    public void process() {
        // if receive a data, print out the data and source thread name
        if (hasInterrupted()) {
            for (int i=0; i<sourceNum; i++) {
                // checks every each channel to see which one has input
                if (in[i].available()>0) {
                    Integer serial = (Integer)in[i].readObject();
                    printInfo("received from:" + in[i].getSourceName()
                        + ":" + "#" + serial.intValue());
                }
            }
        } else { printInfo("time out"); }
    }
}

Register.java:
import jpal.*;
public class Register extends ThreadRegister {
    int sourceNum = 100;
    public Register() {
        super();
        // registers all threads
        ThreadMonitor port = new Port("port");
        register(port);
        ThreadMonitor source[] = new Source[sourceNum];
        for (int i=0; i<sourceNum; i++) {
            String sourceName = "source" + String.valueOf(i);
            source[i] = new Source(sourceName);
            register(source[i]);
        }
    }
    public static void main(String args[]) {
        Register reg = new Register();
        reg.init(args);
        reg.start();
    }
}
```

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley(1977)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley(1999)
3. Lea, Doug: Concurrent Programming in Java, Design Principles and Patterns. Addison-Wesley(1997)
4. Holub, Allen: Programming Java threads in the real world, a Java programmer's guide to threading architectures, Part1-Part9. Java World(September, 1998-May, 1999)
5. Venners, Bill: Inside the Java Virtual Machine. McGraw-Hill(1998)
6. McManis, Chuck: Using Threads in Java, Part2. Java World(May, 1996)