Remote Event Listener

Daxin Cheng
Major Prof: Timothy A. Budd
Minor Prof: Gregg Rothermel
Committee Member: Bruce D’Ambrosio
Department of Computer Science
Oregon State University
February 19, 1998

1 Abstract
Remote Event Listener (REL) is designed to glue remote events and remote listeners dynamically, and dispatch remote events efficiently and transparently for distributed object-oriented systems. Components can be independently developed and remotely interconnected with REL, and software reusability can be improved. Remote Event Listener along with Remote Method Invocation makes distributed system development much easier without knowledge of network communication, and makes formerly developed systems easily extended to distributed systems.

Keywords: remote event listener, system URL, object URL, remote registry, pending registry, remote method invocation, REL Monitor, REL Proxy, REL API, object exportation, object table, registry dispatcher, event dispatcher.

2 Introduction
Object-oriented design and programming is now a dominant approach to developing software systems because OOP enhances software modularity and reusability. Message passing, actually method invocation, is the way that objects interact with each other. An event is a message that occurs unpredictably. If some actions are wired to an event, the actions will be taken whenever the event is fired. This process is called event notification. Event notification is actually achieved by function invocation. We define whatever fires events to be event sources and whoever are interested in events to be event listeners. Formerly, we connect events and listeners in a manner that listeners are decided at object design time, which can not be changed at run time. For more flexibility, events and listeners should be connected and disconnected dynamically at run time. With
this mechanism, components can be designed and developed independently, and interconnected later.

Distributed computing is becoming more and more important due to advanced communication technology, distributed resources and distributed collaborators. In developing distributed systems, we have to take care of communication specifics between components at different locations. Communication peers are usually designed using a Client/Server model and we have to deal with communication protocols, data representation, buffer management and TCP connection management. Clients and servers communicate by the way of request and reply. After sending a request, a client expects to get reply or service from its server in a definite time. Under Client/Server architecture, we are trying to figure out which objects are clients and which objects are servers. But it is often the case there is no sense of clients and servers, and all objects are equally important and cooperate to achieve an objective. For the Juggling Controller system in Fig 1 in the next section where the Juggler starts juggling upon receiving a TimeOut event from the Timer. Consider the Timer and the Juggler, which is client and which is server? If we say the Juggler should be a server because it is listening, the Timer is a client. But what services does the Timer request from the Juggler? There are none. On the other hand, if we regard the Timer as a server and the Juggler as a client, we would have to consider the event registry as a request sent to the server(Timer) and the event notification as service sent back to the client(Juggler). However, events may never be fired. That implies the client may never get reply from the server after a request is made. Actually, without client/server concepts in mind we would be thinking in a more natural way.

Remote Method Invocation(RMI) is a mechanism intended to encapsulate network communication and thus Client/Server architecture, which greatly simplifies distributed system development. With RMI, remote methods can be invoked in a way similar to conventional method invocation, and data can be marshed and unmarshed across network transparently. However, if an object wants its methods to be called remotely, it must be implemented as a remote object, and its corresponding object stub and proxy must be generated to communicate with other objects. This incurs TCP connection delay and performance overhead. For one system, there can be many event sources and listeners. If there is only one common and reusable remote object that is responsible for dealing with communication, event registry and event notification, it would reduce TCP connection overhead and further simplify distributed system development.

Based on RMI, Remote Event Listener(REL) is designed to deal with remote events and listeners more efficiently than RMI. With REL, remote event registries and remote events can be dispatched and processed transparently without knowledge of network communication and remote method invocation. In the following sections, we will introduce object-oriented systems, run-time event/listener connection, distributed object-oriented systems, system and object URL, remote events, remote registries, remote method invocation, remote event listener monitor(REL Monitor), a scenario to show how REL Monitor works, REL Programming API, remote event listener proxy(REL Proxy), lo-
Table 1: Event and Listener Connection for JugglingController

<table>
<thead>
<tr>
<th>event sources</th>
<th>events</th>
<th>listeners</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>TimeOut</td>
<td>Juggler</td>
<td>startJuggling</td>
</tr>
<tr>
<td>Timer</td>
<td>Stop</td>
<td>Juggler</td>
<td>stopJuggling</td>
</tr>
<tr>
<td>Speed</td>
<td>SpeedChange</td>
<td>Juggler</td>
<td>speedChanged</td>
</tr>
<tr>
<td>Juggler</td>
<td>MinSpeedChange</td>
<td>Speed</td>
<td>minSpeedChanged</td>
</tr>
<tr>
<td>Juggler</td>
<td>MaxSpeedChange</td>
<td>Speed</td>
<td>maxSpeedChanged</td>
</tr>
<tr>
<td>Juggler</td>
<td>Juggle</td>
<td>Meter</td>
<td>juggled</td>
</tr>
</tbody>
</table>

cation transparency, object exportation and object table, registry dispatcher, pending registries and event dispatcher, followed by an example illustrating how to use REL API to build distributed systems with remote events and listeners.

3 Object-Oriented Systems and Event Listeners

An object-oriented system is a system that is composed of a number of objects who cooperate to accomplish the system’s functionalities. Each object has its own properties, methods and events. In Figure 1, the Juggling Controller system has four objects: Timer, Speed, Meter, and Juggler. When the Timer times out, it will fire an TimeOut event and send it to the Juggler. Upon receiving the TimeOut event, the Juggler start juggling. In addition to TimeOut event, the Timer can fire Stop event which causes the Juggler to stop juggling. Each time the Juggler juggles, it fires a Juggle event and send it to the Meter object which indicates the total number of times the Juggler has juggled so far. The Speed object can control how fast the Juggler juggles by firing SpeedChange event. However, the Jugger would like to tell the Speed object how fast it can juggle dynamically by firing MinSpeedChange and MaxSpeedChange events. The Timer is an event source; The Meter is an event listener; and both the Speed and Juggler are event sources and at the same time they are event listeners. Table 1 lists the connections between events and listeners. The following pseudocode briefly describes their class definitions and glue between events and listeners. aTimer, aJuggler, aSpeed and aMeter are objects for Timer, Juggler, Speed and Meter classes respectively.

class Timer {
    private int timeoutVal;
    private int timeElapsed;

    public void setTimeout(int to) {
        timeoutVal = to;
    }

    public void startTimer(...) {

Figure 1: Juggling Controller

timeElapsed = 0;
}

private void timeOut(...) {
    // fire TimeOut event
    aJuggler.startJuggling(); //Listener: aJuggler
}

public void stop(...) {
    // fire Stop event
    aJuggler.stopJuggling();
}

public void run {
    ...
    update timeElapsed;
    if (timeElapsed == timeoutVal)
        timeout();
}
}

class Speed {
    private int minSpeed, maxSpeed;
private int speed;

public void setSpeed(int newSpeed) {
    if (newSpeed != speed and minSpeed <= newSpeed <= maxSpeed) {
        speed = newSpeed;
        // fire SpeedChange event;
        aJugger.speedChanged(...); // Juggler adjusts speed of juggling
    }
}

// as a listener, methods required

public void minSpeedChanged(...) {
    minSpeed = newMinSpeed;
    call setSpeed(minSpeed) if (minSpeed>speed) to adjust current speed
}

public void maxSpeedChanged(...) {
    maxSpeed = newMaxSpeed;
    call setSpeed(maxSpeed) if (maxSpeed<speed) to adjust current speed
}

class Juggler {
    private void minSpeedChange(...) {
        // fire MinSpeedChange event and notify the Speed object
        aSpeed.minSpeedChanged(...);
    }

    private void maxSpeedChange(...) {
        // fire MaxSpeedChange event and notify the Speed object
        aSpeed.maxSpeedChanged(...);
    }

    // methods as a listener
    public void speedChanged(...) {
        // change juggling speed
    }

    public void startJuggling() {
        // juggling and fire Juggle event
        aMeter.juggled();
    }
}
From the class description above, events and their listeners connections are established at object design time, and cannot be changed at run time. Take the Timer and Juggler objects for instance. The Timer object designates the specific Juggler object as its Timeout and Stop event Listener at design time. However, the Timer cannot add new listeners, or disconnect its listeners at run time. Similarly, at run time, the Juggler cannot ignore the events from the Timer even though the Juggler is not interested in the events any more; those who are interested in the events cannot join as listeners. Most importantly, components cannot be developed independently. At design time, an object has to determine who are its event listeners, and pointers to its listeners must be obtained at design time so that listeners can be notified when events are fired at run time. For the Timer, for instance, the pointer to the Juggler is necessary for the Juggler to be a listener.

4 Run-time Events and Listeners Connection

To overcome drawbacks of event listener design-time connection described in the previous section, a new mechanism is proposed to allow event and listener connections to be changed dynamically at run time. The key point is that, event sources keep a list of current listeners for each type of events and provide public update functions to change current listener lists at run time. When an event is fired, the event source notified all listeners in the corresponding listener list by calling the listeners’ predefined functions. For example, for the Timer which fires timer events, we define:

1. timerListenerList, a vector that holds all its current listeners.
2. addTimerListener and removeTimerListener methods to allow dynamic registry and unregistry of listeners.

In addition, we define an implementation interface for each category of listeners. For instance, we define TimerListener interface for all TimerEvent listeners, and all TimerEvent listeners must implement this interface.

class TimerEvent extends EventObject {
    // event IDs
private int eventID; // TIMEOUT or STOP

class Timer {
    private Vector timerListenerList;

    public synchronized void addTimerListener(TimerListener listener) {
        if (timerListenerList == null) {
            timerListenerList = new Vector();
        }
        timerListenerList.addElement(listener);
    }

    public synchronized void removeTimerListener(TimerListener listener) {
        if (timerListenerList == null) {
            return;
        }
        timerListenerList.removeElement(listener);
    }

    private void fireTimerEvent(int eventID) {
        TimerEvent event = new TimerEvent(this, eventID);
        if (timerListenerList != null && !timerListenerList.isEmpty()) {
            Vector listenerList = (Vector)timerListenerList.clone();
            // ignore listener change when processing current event
            for (int i=0; i<listenerList.size(); i++) {
                TimerListener listener = (TimerListener)listenerList.elementAt(i);
                switch(eventID) {
                    case TimerEvent.TIMEOUT:
                        listener.timeout(event);
                        break;
                    case TimerEvent.STOP:
                        break;
                }
            }
        }
    }
}

interface TimerListener extends EventListener {
    public void timeout(TimerEvent e);
    public void stop(TimerEvent e);
}

public final static int TIMEOUT = 0;
public final static int STOP = 1;

TimerEvent(Object source, int id) {
    super(source);
    eventID = id;
}

public synchronized void addTimerListener(TimerListener listener) {
    if (timerListenerList == null) {
        timerListenerList = new Vector();
    }
    timerListenerList.addElement(listener);
}

public synchronized void removeTimerListener(TimerListener listener) {
    if (timerListenerList == null) {
        return;
    }
    timerListenerList.removeElement(listener);
}

private void fireTimerEvent(int eventID) {
    TimerEvent event = new TimerEvent(this, eventID);
    if (timerListenerList != null && !timerListenerList.isEmpty()) {
        Vector listenerList = (Vector)timerListenerList.clone();
        // ignore listener change when processing current event
        for (int i=0; i<listenerList.size(); i++) {
            TimerListener listener = (TimerListener)listenerList.elementAt(i);
            switch(eventID) {
                case TimerEvent.TIMEOUT:
                    listener.timeout(event);
                    break;
                case TimerEvent.STOP:
                    break;
            }
        }
    }
}
class Juggler implements TimerListener {
    ...
    // TimerListener implementation
    public void timeout(TimerEvent event) {
        startJuggling();
    }
    public void stop(TimeEvent event) {
        stopJuggling();
    }
}

We notice that, in the Juggling Controller system, the Speed fires speedChange events, and the Juggler fires minSpeedChange and maxSpeedChange events. If we define Event class and Listener interface for each variable(property) change event, We would have a lot of similar event types and listener interfaces. Actually, they all are properties, and we can define PropertyChangeEvent and PropertyChangeListener for change events of various properties.

class PropertyChangeEvent extends EventObject {
    private String propertyName;
    private Object oldValue, newValue;

    PropertyChangeEvent(Object source, String name, Object oldVal, Object newVal) {
        super(source);
        propertyName = name;
        oldValue = oldVal;
        newValue = newVal;
    }

    public Object getOldValue() {
        return oldValue;
    }

    public Object getNewValue() {
        return newValue;
    }
}
public String getPropertyName() {  
    return propertyName;
}

interface PropertyChangeListener extends EventListener {  
    public void propertyChange(PropertyChangeEvent e);
}

For convenience, an auxiliary class PropertyChangeSupport is defined for PropertyChangeEvent sources. We omit the implementation because it is similar to the Timer class.

class PropertyChangeSupport {  
    private Object eventSource;  
    private Vector propertyChangeListenerList;

    PropertyChangeSupport(Object source) {  
        eventSource = source;
    }

    public void addPropertyChangeListener(PropertyChangeListener listener) {  
        changeSupport.addListener(listener);
    }
}

We can use the PropertyChangeSupport class to help implement PropertyChangeEvent sources. The Juggler is a TimerListener, and also a PropertyChangeEvent source that notifies its listeners when minSpeed or maxSpeed is changed.

class Juggler implements TimerListener {  
    // as a PropertyChangeEvent source  
    PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {  
        changeSupport.addListener(listener);
    }
}
public void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
}

private void minSpeedChange(Object oldValue, Object newValue) {
    changeSupport.firePropertyChange("minSpeed", oldValue, newValue);
}

private void maxSpeedChange(Object oldValue, Object newValue) {
    changeSupport.firePropertyChange("maxSpeed", oldValue, newValue);
    ...

// TimerListener implementation

public void timeout(TimerEvent event) {
    startJuggling();
}

public void stop(TimeEvent event) {
    stopJuggling();
}
}

There is one thing we need to clarify. When we implement an object as a specific event listener at design-time, we do not specify who are its event sources. It only means that the object can be one of the event listeners of some event sources at run-time, but it is not necessarily to be. The object must call an event source’s add event listener function to register to be a connected listener. From then on, the object will be notified whenever that kind of events is fired at the event source. Similarly, the object can call the event source’s remove event listener method to unregister to be disconnected from that kind of events fired at the event source. Therefore, run-time connection event/listener model differs from design-time connection model in that, it is event listeners, not event sources, who decide whether to be an actually connected listener at any time. Most importantly, as mentioned before, with run-time connection event/listener model, components(event sources, event listeners, or common objects) can be designed and developed independently, and can be connected later on demand. Independent components greatly improve software reusability.

5 Distributed Object-Oriented Systems and Remote Event Listeners

A distributed object-oriented system(DOOS) is a system that distributes its objects over network, and objects at different locations interact with each other
in a well-defined way to accomplish the system's functionalities. In Figure 2, System Foo has 14 objects among which four objects are on MOON, one on EARTH, four on SUN, three on MARS and two on JUPITER. Similarly, the BAZ system has five objects on EARTH, two on MARS and three on JUPITER. An event listener may be interested in events fired at remote sites. The event listener is a remote event listener, and the events are remote events. Here, remote events/listeners mean that they are across process boundary, and they can be on the same machine.

For the Juggling Controller system in Fig 1, if the four objects move to different locations, for example, the Timer on MOON, the Speed on MARS, the Juggler on SUN, and the Meter on EARTH, then the TimerEvent is a remote event as far as its listener Juggler is concerned. Similarly, the Juggler is a remote event listener as far as the TimerEvent is concerned.

Since all objects belong to the same process for a non-distributed object-oriented system, object variables(or pointers) can be used to refer to objects. However, objects can not be referred to by variables across process boundary. Therefore, how to locate an remote object(i.e. object identity) is an issue under distributed environment. Object identity makes it possible for a remote event listener to register for a certain type of remote events at run time. We use Universal Resource Locator(URL) to globally identify objects.

6 System URL and Object URL

Universal Resource Locator(URL) is originally used for World Wide Web(WWW) to uniquely locate a remote resource. For example, URL:

http://www.cs.orst.edu:8888/people.html

locates the file(resource) people.html on machine www.cs.orst.edu. The WWW server port number is 8888(default 80). Similarly we use URLs to globally identify systems and objects. A system URL consists of machine name, port number if not default, and system name. An object URL consists of its system URL and object name. The port number is the system registry server port number, and we will address system registry server in Section 10. System URL and object URL are of the following formats:

System URL: //machineName:portNumber/systemName
Object URL: //machineName:portNumber/systemName:objectName

For default port number 1099, the portNumber part can be omitted just as the default port number 80 for WWW can be omitted. In addition, for local machine, the machineName can be omitted.

For example, the object Timer for the Juggling Controller system in Fig 1 on machine trek.cs.orst.edu can be identified as following URLs(Port number is default: 1099):

Juggling Controller System URL: //trek.cs.orst.edu/JugglingController
Timer Object URL: //trek.cs.orst.edu/JugglingController/timer

11
Figure 2: Distributed Object-Oriented Systems
As a naming convention, system names start with a capitalized letter, and object names start with an uncapsulated letter.

Objects of the SystemURL and ObjectURL classes are passed as arguments of remote functions (we address Remote Method Invocation later), so both classes must implement the Serializable interface.

```java
public class SystemURL implements java.io.Serializable {
    private String host;
    private int port; // system registry server port number
    private String system;

    SystemURL(String url) {
        /*
         * parse a string into a system URL
         * Format: //host:port/system
         * //host/system (default port)
         * //:port/system (local host)
         * /system (local host and default port)
         */
    }

    public int hashCode() { // hashtable support as a key
        return toString().hashCode();
    }

    public String toString() {
        return ("/" + host + ":" + port + "/" + system);
    }
}
```

```java
public class ObjectURL implements java.io.Serializable {
    private SystemURL systemURL;
    private String object;

    public ObjectURL(String url) {
        /*
         * parse a string into an object URL
         * Format: systemURL string:object
         * //host:port/system:object
         * //host/system:object (default port)
         * //:port/system:object (local host)
         * /system:object (local host, default port)
         * object (local host, default port, local system)
         */
    }
}
```
public int hashCode() {   // hashtable support as a key
    return toString().hashCode();
}

public String toString() {
    return (systemURL.toString() + ":") + object);
}

7 Remote Registries and Remote Events

A remote listener must send a remote registry for a certain type of events to
an event source so that the listener can be connected to the events, and send a
remote unregistry to be disconnected. A remote registry or unregistry should
specify its event source URL, event listener URL and listener type. Listener
type implies event type because there is one-to-one correspondence between
event types and listener types. Like SystemURL and ObjectURL, objects of
RemoteRegistry and RemoteUnregistry are passed as arguments of remote func-
tions, so they implement the Serializable interface.

public class RemoteRegistry implements java.io.Serializable {
    private ObjectURL eventSource, listener;
    private String listenerType;  // listener interface name

    // listenerType: listener interface name. ListenerType implies
    // event type.

    public RemoteRegistry(ObjectURL eventSource,
                          ObjectURL listener, String listenerType) {
        /* set data members to the values passed */
    }

    // Serializable support: default implementation can be omitted.
    private void writeObject(ByteArrayOutputStream s) throws IOException {
        s.writeObject(eventSource);
        s.writeObject(listener);
        s.writeObject(listenerType);
    }

    private void readObject(ByteArrayInputStream s) throws IOException {
        s.readObject(eventSource);
        s.readObject(listener);
        s.readObject(listenerType);
    }
}
public class RemoteUnregistry implements java.io.Serializable {
    private ObjectURL eventSource, listener;
    private String listenerType;
    // listenerType: listener interface name

    public RemoteUnregistry(ObjectURL eventSource,
                            ObjectURL listener, String listenerType) {
        /* set data members to the values passed */
    }

    // serializable support: default implementation can be omitted.
    private void writeObject(ObjectOutputStream s) throws IOException {
        s.writeObject(eventSource);
        s.writeObject(listener);
        s.writeObject(listenerType);
    }

    private void readObject(ObjectInputStream s) throws IOException {
        s.readObject(eventSource);
        s.readObject(listener);
        s.readObject(listenerType);
    }
}

Note that RemoteRegistry and RemoteUnregistry class definitions are exactly the same except they have different class names. A remote registry can be represented by a triple: registry(event source URL, event listener URL, listenerType); and a remote unregistry can be represented by a triple: unregistry(event source URL, event listener URL, listenerType). The remote registry and remote unregistry, for example, sent by the Juggler to the Timer in the Juggling Controller system in Fig 1 can be denoted by:
    registry(Timer URL, Juggler URL, TimerEventListener)
    and
    unregistry(Timer URL, Juggler URL, TimerEventListener).

Once remote events and listeners are connected by remote registries, the listeners will be notified whenever the events occur by sending remote events to the listeners. Since remote events are sent across process boundary, they should include information about their event source URL, remote listener URL, action to perform at the remote listener side, and event information(e.g., mouse pointer position when mouse is clicked). A remote event can be denoted by a quadruple: event(event source URL, event listener URL, event, action). The Serializable interface must be implemented because RemoteEvent objects are passed as arguments of remote functions.

import java.util.EventObject;
public class RemoteEvent implements java.io.Serializable{
    private EventObject event;
    private ObjectURL eventSource;
    private ObjectURL listener;
    private String methodName;  // listener's action

    RemoteEvent(EventObject event, ObjectURL eventSource, 
                 ObjectURL listener, String methodName) {
        /* set data members to the value passed */
    }

    // serializable support: default implementation can be omitted.
    private void writeObject(ObjectOutputStream s) throws IOException {
        s.writeObject(event);
        s.writeObject(eventSource)
        s.writeObject(listener);
        s.writeObject(methodName);
    }

    private void readObject(ObjectInputStream s) throws IOException {
        s.readObject(event);
        s.readObject(eventSource)
        s.readObject(listener);
        s.readObject(methodName);
    }
}

8 Remote Method Invocation

Remote Method Invocation (RMI) allows us to call remote functions in the same way as we call in-process functions by encapsulating network communication. In order for functions to be called remotely, an interface which extends Remote must be defined for the remote functions. That is, a remote interface specifies all functions that can be called remotely. Then we define a class which usually extends UnicastRemoteObject and implements the remote interface. All objects passed as arguments or return values of remote functions must be serializable. The following example Calculator illustrates how to define remote interfaces, remote objects and how to use remote method invocation. We assume that remote object Calculator will be running on machine trek.cs.orst.edu and its user Operator will be running on machine fl engr.orst.edu.

    // CalculatorI.java --- The Calculator remote interface

    import java.rmi.*;
    interface CalculatorI extends Remote {
public abstract int add(int a, int b) throws RemoteException;
public abstract int minus(int a, int b) throws RemoteException;
public abstract int multiply(int a, int b) throws RemoteException;
public abstract int dividedBy(int a, int b) throws RemoteException;
}

// Calculator.java --- the Calculator remote interface implementation

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class Calculator extends UnicastRemoteObject
    implements CalculatorI {
    // must implement constructor to throw RemoteException:
    public Calculator() throws RemoteException {
        // super(); // Called automatically
    }

    // Implementation of the remote interface
    public int add(int a, int b) throws RemoteException {
        return a + b;
    }

    public int minus(int a, int b) throws RemoteException {
        return a - b;
    }

    public int multiply(int a, int b) throws RemoteException {
        return a * b;
    }

    public int dividedBy(int a, int b) throws RemoteException {
        return a / b;
    }

    /*
     * Register itself as Calculator in RMI registry server
     */
    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            System.out.println("Hello, World!");
        } catch(Exception e) {
            System.out.println("Exception: "+e.getMessage());
        }
    }
}
Calculator calculator = new Calculator();
Naming.bind("//trek.cs.orst.edu:2005/Calculator", calculator);

System.out.println("I am ready to do calculation");
} catch(Exception e) {
    e.printStackTrace();
}

The following steps should be followed to prepare for running Calculator:

1. on machine trek.cs.orst.edu, start rmiregistry at port number 2005:
   rmiregistry 2005 &

2. compile CalculatorI.java and Calculator.java:
   javac CalculatorI.java Calculator.java

3. generate skeleton and stub for remote object Calculator
   rmic Calculator

4. start Calculator remote object:
   java Calculator

// Operator.java --- use remote object Calculator
import java.rmi.*;
import java.rmi.registry.*;
import java.awt.*;
import java.awt.event.*;

public class Operator {
    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        // look up the remote object Calculator
        try {
            CalculatorI cal = (CalculatorI) Naming.lookup("//trek.cs.orst.edu:2005/Calculator");
            // remote method invocation
            System.out.println("300 + 200 = " + cal.add(300, 200));
            System.out.println("300 - 200 = " + cal.minus(300, 200));
            System.out.println("300 * 200 = " + cal.multiply(300, 200));
            System.out.println("300 / 200 = " + cal.dividedBy(300, 200));
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
On machine fl.engr.orst.edu, compile and run Operator. The output is:

\[
\begin{align*}
300 + 200 &= 500 \\
300 - 200 &= 100 \\
300 \times 200 &= 60000 \\
300 / 200 &= 1
\end{align*}
\]

9 Remote Event Listener Monitor and Programming API

This section and next section overview Remote Event Listener Monitor (RELMonitor) and how it works, and related concepts will be presented separately in subsequent sections. RELMonitor is designed for hiding communication issues and maintaining the same programming model for remote events and listeners run-time connection/disconnection as discussed in Section 4. One RELMonitor is needed for each system at one machine if the system has remote events or remote listeners. Take the Juggling Controller system in Fig 1 for example, we suppose that the Juggler is on machine sun.orst.edu and the Timer is on machine moon.orst.edu, and that they have the same system name Juggling-Controller. The following code shows how the Juggler registers remotely as a TimerEventListener of the Timer:

```java
// Juggler side on machine sun.orst.edu
RELMonitor relMonitor = new RELMonitor("JugglingController");
relMonitor.exportObject(juggler, "juggler");
relMonitor.addRemoteListener(
    "//moon.orst.edu/JugglingController:timer", // event source
    "juggler", // event listener
    "TimerListener"); // listener type
...
// on exit
relMonitor.exit(); // release resources
```

So, the RELMonitor should has the following API (called REL API) for instantiation, object exportation, remote registry/unregistry, RELProxy extension, and resource release:
1. RELMonitor(systemName); // object instantiation
2. exportObject(objectVariable, objectName); // object exportation
3. addRemoteListener(eventSourceURL, listenerURL, listenerType); // remote registry
4. remoteRemoteListener(eventSourceURL, listenerURL, listenerType); // remote unregistry
5. setRELProxy(newRELProxyClassName); // replace default RELProxy
6. exit(); // release resources

In addition to the programming REL API(local interface), The RELMonitor is a remote object with the following remote interface that is invisible to REL API users:

```java
import java.rmi.*;
public interface RELMonitorI extends Remote {
    public void queueRemoteEvent(RemoteEvent event) throws RemoteException;
    public void queueRemoteRegistry(RemoteRegistry registry) throws RemoteException;
    public void queueRemoteRegistry(RemoteUnregistry unregistry) throws RemoteException;
}
```

From the Fig 3, the RELMonitor has a number of components:

- **Object table**: hold object exportation information. each entry is a triple(object, objectName, RELProxy if any).

- **Remote Event Listener Proxy(RELProxy)**: one RELProxy is generated for each event source that has remote listeners at run-time. The RELProxy is the representative listener for all remote listeners.

- **Local Event Dispatcher**: dispatch events originated at local system to remote systems.

- **Remote Event Dispatcher**: dispatch remote events from outside to local listeners.

- **Local Registry Dispatcher**: dispatch registries from local system to remote systems.

- **Remote Registry Dispatcher**: dispatch remote registries to local event sources.
Figure 3: Remote Event Listener Monitor (REL Monitor)
• Pending Remote Registry Queue: if the Remote Registry Dispatcher cannot process a registry because of unknown event source, the registry will be postponed and put into the Pending Remote Registry Queue for later processing.

The RELMonitor for each system registers itself as the representative of its system in the local System Registry Server so that the system can be found remotely. The main data and method members of the RELMonitor class are sketched as follows:

```java
public class RELMonitor
    extends UnicastRemoteObject
    implements RELMonitorI {

    private static String relProxyClassName = "RELProxy";
    private static SystemURL systemURL;

    private static LocalEventDispatcher localEventDispatcher;
    private static RemoteEventDispatcher remoteEventDispatcher;
    private static LocalRegistryDispatcher localRegistryDispatcher;
    private static RemoteRegistryDispatcher remoteRegistryDispatcher;

    private static Hashtable objectTable = new Hashtable();
    private static Hashtable relMonitorCache = new Hashtable();

    public RELMonitor(String system) throws RemoteException {
        /* Assume: system registry server is listening at
default port number 1099.
*/
        this(system, Registry.REGISTRY_PORT);
    }

    public RELMonitor(String system, int port) throws RemoteException {
        /* Assume: system registry server is listening at the
specified port number
register this system: pair(system name and REL monitor)
create event and registry dispatcher threads.
*/
    }

    public void exit() {
        /* usually called before system termination to release system
registry entry, kill threads, and release other resources.
*/
    }

    public void setRELProxy(String proxyClassName) {
```
/* default: REL Proxy class name is RELProxy
   default REL Proxy can be replaced to allow Proxy extension. */
}

public void exportObject(Object object, String name) throws ObjectExportException {
    /* objects use this method to expose themselves to outside
       so that they can be remote event sources or remote
       event listeners. put mapping(name, object) into the object
       table; The object URL is //localMachine:port/localSystem:name;
       process pending remote registries for this object if any. */
}

public void addRemoteListener(ObjectURL source, ObjectURL listener,
                               String listenerType) {
    /* event listeners use this method to send an event registry to
       a remote event source. From then on, the event source and
       listener are connected for the specified event type. */
}

public void removeRemoteListener(ObjectURL source, ObjectURL listener,
                                  String listenerType) {
    /* event listeners use this method to send an event unregistry to
       a remote event source. From then on, the event source and
       listener are disconnected for the specified event type. */
}

private void queueLocalRegistry(RemoteRegistry registry) {
    /* put the registry into the local registry queue */
}

private void queueLocalRegistry(RemoteUnregistry unregistry) {
    /* put the unregistry into the local registry queue */
}

// Remote Methods: queueRemoteRegistry, queueRemoteRemoteEvent
public void queueRemoteRegistry(Object registry) throws RemoteException {
    /* put the remote registry into the remote registry queue
       at the local system */
}
public void queueRemoteRegistry(RemoteUnregistry unregistry) {
    /* put the remote unregistry into the remote registry queue
    at the local system */
}

public void queueRemoteEvent(RemoteEvent e) throws RemoteException {
    /* put the remote event into the remote event queue
    at the local system */
}

public void queueLocalEvent(RemoteEvent e) {
    /* put the local event into the local event queue */
}

public static Method findMethod(Class cls, String methodName,
        int argCount) {
    /* find a method with the specified name and number of arguments
    for the class using object reflection. (utility) */
}

10 A Scenario: How RELMonitor Works

One System Registry Server is running on each machine, and it maintains a
database of systems that has some objects who might be remote event sources or
remote event listeners. The system must be registered with the System Registry
Server at local machine so that remote registries and events can be dispatched
across system boundary. When an RELMonitor is instantiated for a system, the
system will automatically register itself with the local System Registry Server.
Each entry of system registry has two fields: system name and its RELMonitor
as showed in Fig 4. When a system is found in a System Registry Server,
the reference to its RELMonitor(remote object) will be obtained and then the
remote methods of the RELMonitor can be invoked. The default port number
of System Registry Server is 1099. If a different port number
is used, please refer to the end of Section 16 about what should be changed.

Fig 4 shows what is going on inside and outside of RELMonitor when two
systems start up at two different machines. Step 1 through Step 22 is one typical
sequence that remote registries and remote events are dispatched and processed.

1. System S1: Started and registered itself with its local System Registry
<table>
<thead>
<tr>
<th>SystemName</th>
<th>REL Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>RELM1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SystemName</th>
<th>REL Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>RELM2</td>
</tr>
</tbody>
</table>

**Figure 4: A Scenario: How REL Monitor Works**

Machine M1

Machine M2
2. *System S1*: RELMonitor::exportObject(...) was called to expose object A. An object table entry was allocated for object A, but no RELProxy.

3. *System S1*: RELMonitor::exportObject(...) was called to expose object B. An object table entry was allocated for object B, but no RELProxy.

4. *System S1*: RELMonitor::addRemoteListener(...) was called to launch a remote registry(C, A, Ec). After the registry was processed, Object A would be a remote listener of events of type Ec of object C at System S2. The registry was put into the Local Registry Queue.

5. *System S1*: The Local Registry Dispatcher could not find the remote system S2 for the registry(C, A, Ec) because the remote system S2 has not started yet. The registry was put back into the Local Registry Queue for later try.

6. *System S1*: The Local Registry Dispatcher tried to dispatch the registry(C, A, Ec) again, but still could not find the remote system S2 for the registry, and the registry was put back into the Local Registry Queue for further try.


8. *System S1*: The Local Registry Dispatcher found the remote system S2 for the registry(C, A, Ec) and delivered the registry to the Remote Registry Queue at remote target system S2 by calling the remote method RELMonitor::queueRemoteRegistry(...).

9. *System S2*: The Remote Registry Dispatcher did not find the event source C for the registry(C, A, Ec) because object C has not been exported yet. The registry was put into the Pending Remote Registry Queue.

10. *System S2*: RELMonitor::exportObject(...) was called to expose object C and an object table entry was allocated for object C. Since there was a remote registry pending for object C, an RELProxy was generated for object C and the registry was processed.

11. *System S2*: Object C fired event Ec, and the RELProxy of object C was notified.

12. *System S2*: The RELProxy of object C generated one remote event for each remote listener of event Ec, and put it in the Local Event Queue.

13. *System S2*: The Local Event Dispatcher delivered the events in the Local Event Queue to the Remote Event Queue at remote target system S1 one at a time by calling the remote method RELMonitor::queueRemoteEvent(...). In this case, there is only one event(C, A, Ec, action).
14. **System S1:** The Remote Event Dispatcher got the remote event(C, A, Ec, action), found the listener A by looking up the Object Table, and dispatched the event to the listener A. Note: remote events have event source and listener URLs, but can not carry object variables across process boundary.

15. **System S2:** RELMonitor::exportObject(...) was called to expose object D. An object table entry was allocated for object D, but no RELProxy.

16. **System S2:** RELMonitor::addRemoteListener(...) was called to launch a remote registry(B, D, Eb). After the registry was processed, Object D would be a remote listener of events of type Eb of object B at System S1. The registry was put into the Local Registry Queue.

17. **System S2:** The Local Registry Dispatcher delivered the registry(B, D, Eb) to the Remote Registry Queue at remote target system S1 by calling the remote method RELMonitor::queueRemoteRegistry(...).

18. **System S1:** The Remote Registry Dispatcher found the event source B in the Object Table, processed the registry(B, D, Eb) and created a RELProxy for object B.

19. **System S1:** Object B fired event Eb, and the RELProxy of object B was notified.

20. **System S1:** The RELProxy of object B generated one remote event for each remote listener of event Eb, and put it into the Local Event Queue.

21. **System S1:** The Local Event Dispatcher delivered the events in the Local Event Queue to the Remote Event Queue at remote target system S2 one at a time by calling the remote method RELMonitor::queueRemoteEvent(...). In this case, there was only one event(B, D, Eb, action).

22. **System S2:** The Remote Event Dispatcher got the remote event(B, D, Eb, action), found the listener D by looking up the Object Table, and dispatched the event to the listener D.

11 Remote Event Listener Proxy

When an event is fired at an object(event source), the object notifies all its listeners by calling their designated functions. Event notification by function call requires that an event source and its listeners be in the same process. But for distributed object-oriented systems, remote event sources and listeners are across process/machine boundary. To pretend they are all belong to the same process, a remote event listener proxy(RELProxy) is created for each event source that has remote listeners. No matter how many listeners an event source has, it has one and only one RELProxy which represents all its listeners at different locations. Fig 5 shows the conceptual view of RELProxy. The event source has six
remote event listeners that are represented by one remote event listener proxy within the event source's process space. Whenever events are fired at the event source, they are first dispatched to the remote event listener proxy, who then forward the events to all its remote listeners across process/machine boundary. ProxyREL functions as both an event listener and event source. ProxyREL listens to the events from the event source, and at the same time the ProxyREL dispatches the events to remote listeners as if the events are originated from the ProxyREL. As a event source, RELProxy maintains all different listener lists as shown in Fig 6 and provides a public interface for remote event listeners to register or unregister and update listener lists dynamically at run-time. For each type of event listeners, say WhateverEventListener, the event source interface includes both addWhateverEventListener(...) and remoteWhateverEventListener(...) methods.

RELProxy, as a event listener, has the ability to listen to all kinds of possible events and thus implements all event listener interfaces including ComponentListener, ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener, WindowListener, ActionListener, ItemListener, AdjustmentListener, TextListener, PropertyChangeListener, and VetoableChangeListener.

If there is any remote listener that registers for a certain type of events at an event source, the RELProxy for the event source becomes an event listener of the event source. If all its remote event listeners unregister, the RELProxy is disconnected from the event source.

public class RELProxy implements
    ComponentListener, ContainerListener, FocusListener, KeyListener,
    MouseListener, MouseMotionListener, WindowListener,
    ActionListener, ItemListener, AdjustmentListener, TextListener,
    PropertyChangeListener, VetoableChangeListener {

Figure 5: Conceptual View of Remote Event Listener Proxy
Figure 6: Remote Event Listener Proxy
private Object eventSource;
private String eventSourceName;
private ObjectURL eventSourceURL;
private RELMonitor relMonitor;

// event listener lists
private Vector componentListenerList;
private Vector containerListenerList;
private Vector focusListenerList;
private Vector keyListenerList;
private Vector mouseListenerList;
private Vector mouseMotionListenerList;
private Vector windowListenerList;
private Vector actionListenerList;
private Vector itemListListenerList;
private Vector adjustmentListenerList;
private Vector textListenerList;
private Vector propertyChangeListenerList;
private Vector vetoableChangeListenerList;

/*
ProxyREL function as event source for all event listeners.
implement add/remove listener functions for all types of
event listeners. So remote listeners can register/unregister
at run-time
*/

public void addComponentListener(ObjectURL l)
public void removeComponentListener(ObjectURL l)
public void addContainerListener(ObjectURL l)
public void removeContainerListener(ObjectURL l)
public void addFocusListener(ObjectURL l)
public void removeFocusListener(ObjectURL l)
public void addKeyListener(ObjectURL l)
public void removeKeyListener(ObjectURL l)
public void addMouseListener(ObjectURL l)
public void removeMouseListener(ObjectURL l)
public void addMouseMotionListener(ObjectURL l)
public void removeMouseMotionListener(ObjectURL l)
public void addWindowListener(ObjectURL l)
public void removeWindowListener(ObjectURL l)
public void addActionListener(ObjectURL l)
public void removeActionListener(ObjectURL l)
public void addItemListener(ObjectURL l)
public void removeItemListener(ObjectURL l)
public void addAdjustmentListener(ObjectURL l)
public void removeAdjustmentListener(ObjectURL l)
public void addTextListener(ObjectURL l)
public void removeTextListener(ObjectURL l)
public void addPropertyChangeListener(ObjectURL l)
public void removePropertyChangeListener(ObjectURL l)
public void addVetoableChangeListener(ObjectURL l)
public void removeVetoableChangeListener(ObjectURL l)

/*
   ProxyREL functions as an event listener of all kinds and
   implements all event listener interfaces by forwarding
   events to remote event listeners.
*/

   // ActionListener implementation
public void actionPerformed(ActionEvent event)
   // ComponentListener implementation
public void componentResized( ComponentEvent event )
public void componentMoved( ComponentEvent event )
public void componentShown( ComponentEvent event )
public void componentHidden( ComponentEvent event )
   // ContainerListener implementation
public void componentAdded( ContainerEvent event )
public void componentRemoved( ContainerEvent event )
   // FocusListener implementation
public void focusGained( FocusEvent event )
public void focusLost( FocusEvent event )
   // KeyListener implementation
public void keyTyped( KeyEvent event )
public void keyPressed( KeyEvent event )
public void keyReleased( KeyEvent event )
   // MouseListener implementation
public void mouseClicked( MouseEvent event )
public void mousePressed( MouseEvent event )
public void mouseReleased( MouseEvent event )
public void mouseEntered( MouseEvent event )
public void mouseExited( MouseEvent event )
   // MouseMotionListener implementation
public void mouseDragged( MouseEvent event )
public void mouseMoved( MouseEvent event )
   // WindowListener implementation
public void windowOpened( WindowEvent event )
public void windowClosing( WindowEvent event )
public void windowClosed( WindowEvent event )
public void windowIconified( WindowEvent event )
public void windowDeiconified( WindowEvent event )
public void windowActivated(WindowEvent event)
public void windowDeactivated(WindowEvent event)
  // ActionListener implementation
public void actionPerformed(ActionEvent event)
  // ItemListener implementation
public void itemStateChanged(ItemEvent event)
  // AdjustmentListener implementation
public void adjustmentValueChanged(AdjustmentEvent event)
  // TextListener implementation
public void textValueChanged(TextEvent event)
  // PropertyChangeLister implementation
public void propertyChange(PropertyChangeEvent event)
  // VetoableChangeListener implementation
public void vetoableChange(PropertyChangeEvent event)

protected void deliverEvent(EventObject event, Vector eventListenerList, String methodName) {
  /* this method is called by all methods above to generate
   * remote events for all related listeners and put them into
   * the Local Event Queue for the Local Event Dispatcher to dispatch.
   */
}

If we define a new event and corresponding listener interface, we should extend RELProxy class and add functionalities to deal with this new kind of events and listeners. For example, if we define NewEvent and NewEventListener interface, we should define extended RELProxy as:

class NewRELProxy extends RELProxy implements NewEventListener {
  // event source extension
  private Vector newEventListener;
  public void addNewEventListener(NewEventListener l)
  public void removeNewEventListener(NewEventListener l)

  // event listener extension
  NewEventListener interface implementation
}

The Remote Event Listener Monitor(RELMonitor) should be notified that the extended NewRELProxy replaces the default RELProxy by calling setRELProxy(“NewRELProxy”). We discussed RELMonitor in Section 9.

12 Location Transparency

The RELProxy of an event source is representative of all its remote event listeners, and the RELProxy maintains all different remote event listener lists. An event
source has no knowledge of how many listeners it has and who they are. One indirect level makes location transparency. When an event source migrates from one location to another, its remote event listeners do not need to be notified at all while events are still correctly dispatched to them. When a event listener migrates from one location to another, the RELProxy must be notified, but this is transparent to the event source. The event source only knows that the RELProxy is its in-process listener if there is any remote event listeners.

13 Object Exportation and Object Table

A remote registry(sourceURL, listenerURL, listenerType) only has information about object URL for its event source and listener because object variables can not go across process boundary. In order to find the event source object for a remote registry and to find the listener object for a remote event, mapping between objects and their names is necessary for those remote event sources and listeners. This mapping is called Object Table as in Fig 4. Each entry of the Object Table has three fields: object, name, and RELProxy. If the object does not have remote listeners, the RELProxy field will be null.

Before an object can be referred to remotely by one object URL, the object must announce to the outside world what’s its name. This process is called object exportation. Actually, the process of object exportation is to allocate one entry from the Object Table and fill out the information. After exportation the object’s URL is automatically formed as //machine:port/system:objectName. Obviously, all exported objects from one system at one machine must have different exported names.

14 Registry Dispatcher and Pending Registries

The Local and Remote Registry Dispatchers are both threads that, usually in the sleep state, wake up whenever there are registries/unregistries in their registry queues, and then fall asleep after dispatching the registries/unregistries. They act the same way as the Local and Remote Event Dispatchers that will be addressed in the next section.

The Local Registry Dispatcher is responsible for dispatching the registries and unregistries in Local Registry Queue at its local system to the Remote Registry Queue of its target system by calling the remote function:

```
RELMonitor:queueRemoteRegistry(RemoteRegistry)
```

If the target system for a registry can not be found, the dispatcher will poll the remote system later by putting the registry back to the end of the Local Registry Queue.

The Remote Registry Dispatcher is responsible for dispatching registries and unregistries requests from remote systems(i.e. registries and unregistries in the Remote Registry Queue). If the event source for a registry has already been exported and can be found in the Object Table, the dispatcher will
1. generate a RELProxy for the event source if the RELProxy does not exist,
2. update the corresponding remote listener list in the RELProxy,
3. register the RELProxy to be a listener of the type specified by the remote
   registry if the RELProxy is not the type of listener of the event source.

For unregistries, only Step 2 is taken.

On the other hand, if the event source object has not been exported yet,
the Remote Registry Dispatcher will move the registry from the Remote Regi-
istry Queue to the Pending Remote Registry Queue so that other registries or
unregistries can be processed. Once the event source for the pending registry
is exported, the registry will be removed from the Pending Remote Registry
Queue and processed. An unregistry will be ignored if its event source can not
be found.

```java
public class LocalRegistryDispatcher extends Thread {
    public synchronized Object getRegistry() {
        /*
        * get a registry/unregistry from the local registry queue
        * and blocked if the queue is empty
        */
    }

    public synchronized void putRegistry(Object registry) {
        /* put a registry/unregistry into the local registry queue */
    }

    public void run() {
        while (true) {
            Object registry = getRegistry();
            if (registry instanceof RemoteRegistry)
                dispatchLocalRegistry((RemoteRegistry)registry);
            else
                dispatchLocalUnregistry((RemoteUnregistry)registry);
        }
    }

    private void dispatchLocalRegistry(RemoteRegistry remoteRegistry) {
        /*
        * look up the system registry at the target(event source) machine
        * for the target system. If the system can not be found, the
        * registry will be put back at the end of the Local Registry Queue for
        * further retry. If the system is found, the registry will be delivered
        * to the target Remote Registry Queue by remote method invocation.
        * We use system RELMonitor cache to improve performance.
        */
    }
```
private void dispatchLocalUnregistry(RemoteUnregistry remoteUnregistry) {
    /*
     * Similar to dispatchLocalRegistry. The target system RELMonitor should
     * be found in the cache, other it will be ignored.
     */
}

public class RemoteRegistryDispatcher extends Thread {
    public void processPendingRegistry(String objectName) {
        /*
         * process all the pending registry for the object if any *
         */
    }
    public synchronized Object getRegistry() {
        /*
         * get a registry/unregistry from the remote registry queue
         * and blocked if the queue is empty
         */
    }
    public synchronized void putRegistry(Object registry) {
        /*
         * put a registry/unregistry into the remote registry queue *
         */
    }
    public void run() {
        while (true) {
            Object registry = getRegistry();
            if (registry instanceof RemoteRegistry)
                dispatchRemoteRegistry(((RemoteRegistry)registry));
            else
                dispatchRemoteUnregistry(((RemoteUnregistry)registry));
        }
    }
    private void dispatchRemoteRegistry(RemoteRegistry remoteRegistry) {
        /*
         * try to find the event source at the local system for the registry.
         * if the event source has not been exported, put the registry into
         * the Pending Remote Registry Queue. If the event source has been
         * exported(in the object table), but the RELProxy has not been
         * created, that is, this registry is the first registry for the object,
         * then create the RELProxy and register it as the listener of the
         * event source. If the RELProxy is already created, just add the
         * remote listener into the corresponding listener list in the RELProxy.
         */
    }
}

public class RemoteRegistryDispatcher extends Thread {
    public void processPendingRegistry(String objectName) {
        /*
         * process all the pending registry for the object if any *
         */
    }
    public synchronized Object getRegistry() {
        /*
         * get a registry/unregistry from the remote registry queue
         * and blocked if the queue is empty
         */
    }
    public synchronized void putRegistry(Object registry) {
        /*
         * put a registry/unregistry into the remote registry queue *
         */
    }
    public void run() {
        while (true) {
            Object registry = getRegistry();
            if (registry instanceof RemoteRegistry)
                dispatchRemoteRegistry(((RemoteRegistry)registry));
            else
                dispatchRemoteUnregistry(((RemoteUnregistry)registry));
        }
    }
    private void dispatchRemoteRegistry(RemoteRegistry remoteRegistry) {
        /*
         * try to find the event source at the local system for the registry.
         * if the event source has not been exported, put the registry into
         * the Pending Remote Registry Queue. If the event source has been
         * exported(in the object table), but the RELProxy has not been
         * created, that is, this registry is the first registry for the object,
         * then create the RELProxy and register it as the listener of the
         * event source. If the RELProxy is already created, just add the
         * remote listener into the corresponding listener list in the RELProxy.
         */
    }
}
private void dispatchRemoteUnregistry(RemoteUnregistry remoteUnregistry) {
    /*
     * Similar to dispatchRemoteRegistry. Remote the listener from the listener list in the RELProxy.
     */
}

15 Event Dispatcher

The Local Event Dispatcher dispatches events from the local system to the Remote Event Queue of the target system by calling the remote method:

```
RELMonitor:queueRemoteEvent(RemoteEvent)
```

The Remote Event Dispatcher gets one event(RemoteEvent) at a time from the Remote Event Queue, and delivers it to its listener at the local system by calling the method specified by the RemoteEvent. The method specified by RemoteEvent objects are method names, so object reflection is used to get and invoke the method.

```java
public class LocalEventDispatcher extends Thread {
    public synchronized RemoteEvent getEvent() {
        /*
         * get an event from the local event queue and blocked if the queue is empty.
         */
    }

    public synchronized void putEvent(RemoteEvent event) {
        /* put an event into the local event queue */
    }

    public void run() {
        while (true) {
            RemoteEvent event = getEvent();
            dispatchLocalEvent(event);
        }
    }

    private void dispatchLocalEvent(RemoteEvent remoteEvent) {
        /*
         * find the RELMonitor of the target listener system and deliver the event to the remote event queue at the target system by remote method invocation. We use RELMonitor cache to improve
         */
    }
```
public class RemoteEventDispatcher extends Thread {
    public synchronized RemoteEvent getEvent() {
        /* get an event from the remote event queue and blocked if the queue is empty. */
    }

    public synchronized void putEvent(RemoteEvent event) {
        /* put an event into the remote event queue */
    }

    public void run() {
        while (true) {
            RemoteEvent event = getEvent();

            try {
                dispatchRemoteEvent(event);
            } catch (ObjectExportException e) {
                e.printStackTrace();
            }
        }
    }

    private void dispatchRemoteEvent(RemoteEvent remoteEvent)
        throws ObjectExportException {
        /* find the listener at the local system and get the method specified as a string in remoteEvent by object reflection. Finally call the method(listener action). */
    }
}

16 Example

We use a very simple example to demonstrate how to use REL API to wire remote events and listeners. We have four systems: three Account Operators(on jasper.cs.orst.edu, snake.cs.orst.edu, and kite.cs.orst.edu), and one Account on
If the Incr(+10) button is pressed at any of the three AccountOperator systems, the balance at trek.cs.orst.edu will increase by 10. Similarly if the Decr(-5) button is pressed, the balance will decrease by 5. Whenever the balance is changed at trek.cs.orst.edu, they will be reflected at all the three AccountOperators. Table 2 shows the remote event and listener connections.

```java
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

class BalanceData extends TextField implements ActionListener {
    int value = 0;
    PropertyChangeSupport propertyChangeSupport;

    BalanceData(int n) {
        propertyChangeSupport = new PropertyChangeSupport(this);
        setValue(n);
    }
}```
Table 2: Remote Event and listener connections

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Event Listener</th>
<th>Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incr(+10) at jasper</td>
<td>Balance at trek</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Decr(-5) at jasper</td>
<td>Balance at trek</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Incr(+10) at snake</td>
<td>Balance at trek</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Decr(-5) at snake</td>
<td>Balance at trek</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Incr(+10) at kite</td>
<td>Balance at trek</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Decr(-5) at kite</td>
<td>Balance at trek</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Balance at trek</td>
<td>Balance at jasper</td>
<td>PropertyChangeEvent</td>
</tr>
<tr>
<td>Balance at trek</td>
<td>Balance at snake</td>
<td>PropertyChangeEvent</td>
</tr>
<tr>
<td>Balance at trek</td>
<td>Balance at kite</td>
<td>PropertyChangeEvent</td>
</tr>
</tbody>
</table>

```java
public void addPropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeSupport.removePropertyChangeListener(listener);
}

private void setValue(int newValue) {
    if (newValue != value) {
        propertyChangeSupport.firePropertyChange("value",
            new Integer(value), new Integer(newValue));
        value = newValue;
        setText("" + value);
    }
}

public void actionPerformed(ActionEvent e) {
    // for local events
}

public void actionPerformed(ActionEvent e, ObjectURL eventSource) {
    String source = eventSource.getObject();
    if (source.equals("incr(+10)") {
        setValue(value + 10);
    } else if (source.equals("decr(-5)") {
        setValue(value - 5);
    }
}
```
class Account extends Frame {
  private static String[] accountOperatorHost;
  RELMonitor relm;

  private BalanceData balance;

  Account() {
    super("Account");
    Panel p = new Panel(new FlowLayout());
    Label label = new Label("Balance:");
    balance = new BalanceData(500);
    p.add(label);
    p.add(balance);
    add(p);
    try {
      relm = new RELMonitor("Account");
      relm.exportObject(balance, "balance");
      catch (Exception e) {
        e.printStackTrace();
      }
      new WindowCloser(this, relm);
      ObjectURL listener = new ObjectURL("balance");
      for (int i=0; i<accountOperatorHost.length; i++) {
        ObjectURL eventSource = new ObjectURL("/\" + accountOperatorHost[i]
          + "/AccountOperator:incr\ (+10)\")
          + "/AccountOperator:decr\ (-5)\";
        relm.addRemoteListener(eventSource, listener, "ActionListener");
        eventSource = new ObjectURL("/\" + accountOperatorHost[i]
          + "/AccountOperator:incr\ (+10)\")
          + "/AccountOperator:decr\ (-5)\";
        relm.addRemoteListener(eventSource, listener, "ActionListener");
      }
    }
  }

  public static void main(String argv[]) {
    if (argv.length == 0) {
      System.out.println("Syntax: java Account AccountOperatorHosts");
      return;
    }
    accountOperatorHost = argv;
    Account account = new Account();
    account.setSize(200, 200);
    account.show();
  }
}
class BalanceView extends TextField implements PropertyChangeListener {
    BalanceView() {
        setText("***");
    }

    public void propertyChange(PropertyChangeEvent event) {
        // local event
    }

    public void propertyChange(PropertyChangeEvent event, ObjectURL eventSource) {
        setText(""+(Integer)event.getNewValue().intValue());
    }
}

class AccountOperator extends Frame {
    private static String accountHost;
    private RELMonitor relm;

    private BalanceView balance;

    AccountOperator() {
        super("Account Operator");
        Panel p = new Panel(new FlowLayout());
        Label label = new Label("Balance:");
        balance = new BalanceView();

        Button incr = new Button("Incr(+10)");
        Button decr = new Button("Decr(-5)");
        p.add(label);
        p.add(balance);
        p.add(incr);
        p.add(decr);
        add(p);
        try {
            relm = new RELMonitor("AccountOperator");
            relm.exportObject(incr, "incr(+10)");
            relm.exportObject(decr, "decr(-5)");
            relm.exportObject(balance, "balance");
try {
    new WindowCloser(this, relm);
    ObjectURL listener = new ObjectURL("balance");
    ObjectURL eventSource = new ObjectURL("/Account:balance");
    relm.addRemoteListener(eventSource, listener, "PropertyChangeListener");
}

public static void main(String argv[]) {
    if (argv.length == 0) {
        System.out.println("Syntax: java AccountOperator AccountHost");
        return;
    }
    accountHost = argv[0];
    AccountOperator ao = new AccountOperator();
    ao.setSize(200, 200);
    ao.show();
}

------- Utility--------
/*
 * WindowCloser.java
 * WindowCloser is a utility class that can be used by applications to remove their registry entries from the system registry server when they terminate.
 */
import java.awt.*;
import java.awt.event.*;

public class WindowCloser implements WindowListener {
    private RELMonitor relMonitor;

    public WindowCloser(Window window, RELMonitor relm) {
        relMonitor = relm;
        window.addWindowListener(this);
    }

    public void windowOpened(WindowEvent e) {
    }

    public void windowClosing(WindowEvent e) {
}
relMonitor.exit();
System.exit(0);
}

public void windowClosed(WindowEvent e) {
    relMonitor.exit();
    System.exit(0);
}

public void windowIconified(WindowEvent e) {
}

public void windowDeiconified(WindowEvent e) {
}

public void windowActivated(WindowEvent e) {
}

public void windowDeactivated(WindowEvent e) {
}
}

The procedure to start the Account and the three AccountOperators:

1. One system registry server for one machine. Start system registry server on each machine at port number 1099 if necessary:

    sysregistry 1099 &

If the default port number 1099 has already been used by other applications, select another available port number, say portNumber. In this case, the constructor RELMonitor(systemName, portNumber) should be used instead of RELMonitor(systemName) in instantiating the RELMonitor class.


3. Start Account(on trek) and AccountOperators(on jasper, snake, and kite). The order of starting systems is not important because Remote Event Listener can deal with pending registries and thus registries will never be lost.

17 Conclusion

Remote Method Invocation and Remote Event Listener make distributed system development much more easier by encapsulating network communication, and we can concentrate on achieving system functionalities rather than on dealing
with communication details. Remote Event Listener is designed to register and dispatch remote events more efficiently. From previous discussion, a formerly developed non-distributed system can be easily transformed into distributed one. Local event listeners become remote event listeners, and REL Programming API can be used to wire remote events and remote listeners, and dispatch remote events. Local method invocations become remote method invocations if callers and their callees are separated across process boundary. In this case, the callees should be implemented as remote objects, and arguments and return values for remote functions need to be implemented as Serializable. Network communication for distributed systems is completely transparent to system developers.