ZooKeeper Java Example

by

Table of contents

1 A Simple Watch Client.....2

1.1 Requirements.....2

1.2 Program Design.....2

2 The Executor Class.....2

3 The DataMonitor Class.....5

4 Complete Source Listings.....7

1 A Simple Watch Client

To introduce you to the ZooKeeper Java API, we develop here a very simple watch client. This ZooKeeper client watches a ZooKeeper node for changes and responds to by starting or stopping a program.

1.1 Requirements

The client has four requirements:

- It takes as parameters:
  - the address of the ZooKeeper service
  - the name of a znode - the one to be watched
  - the name of a file to write the output to
  - an executable with arguments.
- It fetches the data associated with the znode and starts the executable.
- If the znode changes, the client refetches the contents and restarts the executable.
- If the znode disappears, the client kills the executable.

1.2 Program Design

Conventionally, ZooKeeper applications are broken into two units, one which maintains the connection, and the other which monitors data. In this application, the class called the **Executor** maintains the ZooKeeper connection, and the class called the **DataMonitor** monitors the data in the ZooKeeper tree. Also, Executor contains the main thread and contains the execution logic. It is responsible for what little user interaction there is, as well as interaction with the exectuable program you pass in as an argument and which the sample (per the requirements) shuts down and restarts, according to the state of the znode.

2 The Executor Class

The Executor object is the primary container of the sample application. It contains both the **ZooKeeper** object, **DataMonitor**, as described above in Program Design.

```
// from the Executor class...

public static void main(String[] args) {
    if (args.length < 4) {
        System.err
                .println("USAGE: Executor hostPort znode filename program [args ...]");
        System.exit(2);
    }
    String hostPort = args[0];
    String znode = args[1];
    String filename = args[2];
    String exec[] = new String[args.length - 3];
```

```
        System.arraycopy(args, 3, exec, 0, exec.length);
        try {
            new Executor(hostPort, znode, filename, exec).run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public Executor(String hostPort, String znode, String filename,
            String exec[]) throws KeeperException, IOException {
        this.filename = filename;
        this.exec = exec;
        zk = new ZooKeeper(hostPort, 3000, this);
        dm = new DataMonitor(zk, znode, null, this);
    }

    public void run() {
        try {
            synchronized (this) {
                while (!dm.dead) {
                    wait();
                }
            }
        } catch (InterruptedException e) {
        }
    }
```

Recall that the Executor's job is to start and stop the executable whose name you pass in on the command line. It does this in response to events fired by the ZooKeeper object. As you can see in the code above, the Executor passes a reference to itself as the Watcher argument in the ZooKeeper constructor. It also passes a reference to itself as DataMonitorListener argument to the DataMonitor constructor. Per the Executor's definition, it implements both these interfaces:

```
public class Executor implements Watcher, Runnable, DataMonitor.DataMonitorListener {
...
```

The **Watcher** interface is defined by the ZooKeeper Java API. ZooKeeper uses it to communicate back to its container. It supports only one method, `process()`, and ZooKeeper uses it to communciates generic events that the main thread would be intersted in, such as the state of the ZooKeeper connection or the ZooKeeper session.The Executor in this example simply forwards those events down to the DataMonitor to decide what to do with them. It does this simply to illustrate the point that, by convention, the Executor or some Executor-like object "owns" the ZooKeeper connection, but it is free to delegate the events to other events to other objects. It also uses this as the default channel on which to fire watch events. (More on this later.)

```
    public void process(WatchedEvent event) {
        dm.process(event);
```

```
    }
```

The **DataMonitorListener** interface, on the other hand, is not part of the the ZooKeeper API. It is a completely custom interface, designed for this sample application. The DataMonitor object uses it to communicate back to its container, which is also the the Executor object.The DataMonitorListener interface looks like this:

```
public interface DataMonitorListener {
    /**
    * The existence status of the node has changed.
    */
    void exists(byte data[]);

    /**
    * The ZooKeeper session is no longer valid.
    *
    * @param rc
    * the ZooKeeper reason code
    */
    void closing(int rc);
}
```

This interface is defined in the DataMonitor class and implemented in the Executor class. When `Executor.exists()` is invoked, the Executor decides whether to start up or shut down per the requirements. Recall that the requires say to kill the executable when the znode ceases to *exist*.

When `Executor.closing()` is invoked, the Executor decides whether or not to shut itself down in response to the ZooKeeper connection permanently disappearing.

As you might have guessed, DataMonitor is the object that invokes these methods, in response to changes in ZooKeeper's state.

Here are Executor's implementation of `DataMonitorListener.exists()` and `DataMonitorListener.closing`:

```
public void exists( byte[] data ) {
    if (data == null) {
        if (child != null) {
            System.out.println("Killing process");
            child.destroy();
            try {
                child.waitFor();
            } catch (InterruptedException e) {
            }
        }
        child = null;
    } else {
        if (child != null) {
            System.out.println("Stopping child");
            child.destroy();
            try {
```

```
                child.waitFor();
            } catch (InterruptedException e) {
            e.printStackTrace();
            }
        }
        try {
            FileOutputStream fos = new FileOutputStream(filename);
            fos.write(data);
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            System.out.println("Starting child");
            child = Runtime.getRuntime().exec(exec);
            new StreamWriter(child.getInputStream(), System.out);
            new StreamWriter(child.getErrorStream(), System.err);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public void closing(int rc) {
    synchronized (this) {
        notifyAll();
    }
}
```

3 The DataMonitor Class

The DataMonitor class has the meat of the ZooKeeper logic. It is mostly asynchronous and event driven. DataMonitor kicks things off in the constructor with:

```
public DataMonitor(ZooKeeper zk, String znode, Watcher chainedWatcher,
        DataMonitorListener listener) {
    this.zk = zk;
    this.znode = znode;
    this.chainedWatcher = chainedWatcher;
    this.listener = listener;

    // Get things started by checking if the node exists. We are going
    // to be completely event driven
    zk.exists(znode, true, this, null);
}
```

The call to `ZooKeeper.exists()` checks for the existence of the znode, sets a watch, and passes a reference to itself (`this`) as the completion callback object. In this sense, it kicks things off, since the real processing happens when the watch is triggered.

> **Note:**
>
> Don't confuse the completion callback with the watch callback. The `ZooKeeper.exists()` completion callback, which happens to be the method `StatCallback.processResult()`

implemented in the DataMonitor object, is invoked when the asynchronous *setting of the watch* operation (by `ZooKeeper.exists()`) completes on the server.

The triggering of the watch, on the other hand, sends an event to the *Executor* object, since the Executor registered as the Watcher of the ZooKeeper object.

As an aside, you might note that the DataMonitor could also register itself as the Watcher for this particular watch event. This is new to ZooKeeper 3.0.0 (the support of multiple Watchers). In this example, however, DataMonitor does not register as the Watcher.

When the `ZooKeeper.exists()` operation completes on the server, the ZooKeeper API invokes this completion callback on the client:

```
public void processResult(int rc, String path, Object ctx, Stat stat) {
    boolean exists;
    switch (rc) {
    case Code.Ok:
        exists = true;
        break;
    case Code.NoNode:
        exists = false;
        break;
    case Code.SessionExpired:
    case Code.NoAuth:
        dead = true;
        listener.closing(rc);
        return;
    default:
        // Retry errors
        zk.exists(znode, true, this, null);
        return;
    }

    byte b[] = null;
    if (exists) {
        try {
            b = zk.getData(znode, false, null);
        } catch (KeeperException e) {
            // We don't need to worry about recovering now. The watch
            // callbacks will kick off any exception handling
            e.printStackTrace();
        } catch (InterruptedException e) {
            return;
        }
    }
    if ((b == null && b != prevData)
            || (b != null && !Arrays.equals(prevData, b))) {
        listener.exists(b);
        prevData = b;
    }
}
```

The code first checks the error codes for znode existence, fatal errors, and recoverable errors. If the file (or znode) exists, it gets the data from the znode, and then invoke the exists()

callback of Executor if the state has changed. Note, it doesn't have to do any Exception processing for the getData call because it has watches pending for anything that could cause an error: if the node is deleted before it calls `ZooKeeper.getData()`, the watch event set by the `ZooKeeper.exists()` triggers a callback; if there is a communication error, a connection watch event fires when the connection comes back up.

Finally, notice how DataMonitor processes watch events:

```
public void process(WatchedEvent event) {
    String path = event.getPath();
    if (event.getType() == Event.EventType.None) {
        // We are are being told that the state of the
        // connection has changed
        switch (event.getState()) {
        case SyncConnected:
            // In this particular example we don't need to do anything
            // here - watches are automatically re-registered with
            // server and any watches triggered while the client was
            // disconnected will be delivered (in order of course)
            break;
        case Expired:
            // It's all over
            dead = true;
            listener.closing(KeeperException.Code.SessionExpired);
            break;
        }
    } else {
        if (path != null && path.equals(znode)) {
            // Something has changed on the node, let's find out
            zk.exists(znode, true, this, null);
        }
    }
    if (chainedWatcher != null) {
        chainedWatcher.process(event);
    }
}
```

If the client-side ZooKeeper libraries can re-establish the communication channel (SyncConnected event) to ZooKeeper before session expiration (Expired event) all of the session's watches will automatically be re-established with the server (auto-reset of watches is new in ZooKeeper 3.0.0). See ZooKeeper Watches in the programmer guide for more on this. A bit lower down in this function, when DataMonitor gets an event for a znode, it calls `ZooKeeper.exists()` to find out what has changed.

4 Complete Source Listings

Executor.javaExecutor.java

```
/**
 * A simple example program to use DataMonitor to start and
 * stop executables based on a znode. The program watches the
```

```
 * specified znode and saves the data that corresponds to the
 * znode in the filesystem. It also starts the specified program
 * with the specified arguments when the znode exists and kills
 * the program if the znode goes away.
 */
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;

public class Executor
    implements Watcher, Runnable, DataMonitor.DataMonitorListener
{
    String znode;

    DataMonitor dm;

    ZooKeeper zk;

    String filename;

    String exec[];

    Process child;

    public Executor(String hostPort, String znode, String filename,
            String exec[]) throws KeeperException, IOException {
        this.filename = filename;
        this.exec = exec;
        zk = new ZooKeeper(hostPort, 3000, this);
        dm = new DataMonitor(zk, znode, null, this);
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length < 4) {
            System.err
                    .println("USAGE: Executor hostPort znode filename program
 [args ...]");
            System.exit(2);
        }
        String hostPort = args[0];
        String znode = args[1];
        String filename = args[2];
        String exec[] = new String[args.length - 3];
        System.arraycopy(args, 3, exec, 0, exec.length);
        try {
            new Executor(hostPort, znode, filename, exec).run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    /******************************************************************************
     * We do process any events ourselves, we just need to forward them on.
     *
     * @see
org.apache.zookeeper.Watcher#process(org.apache.zookeeper.proto.WatcherEvent)
     */
    public void process(WatchedEvent event) {
        dm.process(event);
    }

    public void run() {
        try {
            synchronized (this) {
                while (!dm.dead) {
                    wait();
                }
            }
        } catch (InterruptedException e) {
        }
    }

    public void closing(int rc) {
        synchronized (this) {
            notifyAll();
        }
    }

    static class StreamWriter extends Thread {
        OutputStream os;

        InputStream is;

        StreamWriter(InputStream is, OutputStream os) {
            this.is = is;
            this.os = os;
            start();
        }

        public void run() {
            byte b[] = new byte[80];
            int rc;
            try {
                while ((rc = is.read(b)) > 0) {
                    os.write(b, 0, rc);
                }
            } catch (IOException e) {
            }

        }
    }

    public void exists(byte[] data) {
        if (data == null) {
            if (child != null) {
                System.out.println("Killing process");
                child.destroy();
                try {
                    child.waitFor();
                } catch (InterruptedException e) {
                }
```

```
            }
            child = null;
        } else {
            if (child != null) {
                System.out.println("Stopping child");
                child.destroy();
                try {
                    child.waitFor();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
                FileOutputStream fos = new FileOutputStream(filename);
                fos.write(data);
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                System.out.println("Starting child");
                child = Runtime.getRuntime().exec(exec);
                new StreamWriter(child.getInputStream(), System.out);
                new StreamWriter(child.getErrorStream(), System.err);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

DataMonitor.javaDataMonitor.java

```
/**
 * A simple class that monitors the data and existence of a ZooKeeper
 * node. It uses asynchronous ZooKeeper APIs.
 */
import java.util.Arrays;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.AsyncCallback.StatCallback;
import org.apache.zookeeper.KeeperException.Code;
import org.apache.zookeeper.data.Stat;

public class DataMonitor implements Watcher, StatCallback {

    ZooKeeper zk;

    String znode;

    Watcher chainedWatcher;

    boolean dead;
```

```
    DataMonitorListener listener;

    byte prevData[];

    public DataMonitor(ZooKeeper zk, String znode, Watcher chainedWatcher,
            DataMonitorListener listener) {
        this.zk = zk;
        this.znode = znode;
        this.chainedWatcher = chainedWatcher;
        this.listener = listener;
        // Get things started by checking if the node exists. We are going
        // to be completely event driven
        zk.exists(znode, true, this, null);
    }

    /**
     * Other classes use the DataMonitor by implementing this method
     */
    public interface DataMonitorListener {
        /**
         * The existence status of the node has changed.
         */
        void exists(byte data[]);

        /**
         * The ZooKeeper session is no longer valid.
         *
         * @param rc
         *                the ZooKeeper reason code
         */
        void closing(int rc);
    }

    public void process(WatchedEvent event) {
        String path = event.getPath();
        if (event.getType() == Event.EventType.None) {
            // We are are being told that the state of the
            // connection has changed
            switch (event.getState()) {
            case SyncConnected:
                // In this particular example we don't need to do anything
                // here - watches are automatically re-registered with
                // server and any watches triggered while the client was
                // disconnected will be delivered (in order of course)
                break;
            case Expired:
                // It's all over
                dead = true;
                listener.closing(KeeperException.Code.SessionExpired);
                break;
            }
        } else {
            if (path != null && path.equals(znode)) {
                // Something has changed on the node, let's find out
                zk.exists(znode, true, this, null);
            }
        }
        if (chainedWatcher != null) {
            chainedWatcher.process(event);
        }
```

```
    }

    public void processResult(int rc, String path, Object ctx, Stat stat) {
        boolean exists;
        switch (rc) {
        case Code.Ok:
            exists = true;
            break;
        case Code.NoNode:
            exists = false;
            break;
        case Code.SessionExpired:
        case Code.NoAuth:
            dead = true;
            listener.closing(rc);
            return;
        default:
            // Retry errors
            zk.exists(znode, true, this, null);
            return;
        }

        byte b[] = null;
        if (exists) {
            try {
                b = zk.getData(znode, false, null);
            } catch (KeeperException e) {
                // We don't need to worry about recovering now. The watch
                // callbacks will kick off any exception handling
                e.printStackTrace();
            } catch (InterruptedException e) {
                return;
            }
        }
        if ((b == null && b != prevData)
                || (b != null && !Arrays.equals(prevData, b))) {
            listener.exists(b);
            prevData = b;
        }
    }
}
```