Tutorial

Programming with ZooKeeper - A quick tutorial

In this tutorial, we show simple implementations of barriers and producer-consumer queues using !ZooKeeper. We call the respective classes Barrier and Queue. These examples assume that you have at least one !ZooKeeper server running.

Both primitives use the following common excerpt of code:

```
static ZooKeeper zk = null;
static final Object mutex = new Object();

String root;

SyncPrimitive(String address)
throws KeeperException, IOException {
    if(zk == null){
            System.out.println("Starting ZK:");
            zk = new ZooKeeper(address, 3000, this);
            System.out.println("Finished starting ZK: " + zk);
    }
}

public void process(WatcherEvent event) {
    synchronized (mutex) {
        mutex.notify();
    }
}
```

Both classes extend !SyncPrimitive. In this way, we execute steps that are common to all primitives in the constructor of !SyncPrimitive. To keep the examples simple, we create a !ZooKeeper object the first time we instantiate either a barrier object or a queue object, and we declare a static variable that is a reference to this object. The subsequent instances of Barrier and Queue check whether a !ZooKeeper object exists. Alternatively, we could have the application creating a !ZooKeeper object and passing it to the constructor of Barrier and Queue.

We use the process() method to process notifications triggered due to watches. In the following discussion, we present code that sets watches. A watch is internal structure that enables !ZooKeeper to notify a client of a change to a node. For example, if a client is waiting for other clients to leave a barrier, then it can set a watch and wait for modifications to a particular node, which can indicate that it is the end of the wait. This point becomes clear once we go over the examples.

Barriers

A barrier is a primitive that enables a group of processes to synchronize the beginning and the end of a computation. The general idea of this implementation is to have a barrier node that serves the purpose of being a parent for individual process nodes. Suppose that we call the barrier node " /b1". Each process "p" then creates a node "/b1/p". Once enough processes have created their corresponding nodes, joined processes can start the computation.

In this example, each process instantiates a Barrier object, and its constructor takes as parameters:

-
  -
    -
      - The address of a !ZooKeeper server (e.g., "zoo1.foo.com:2181");
      - The path of the barrier node on !ZooKeeper (e.g., "/b1");
      - The size of the group of processes.

The constructor of Barrier passes the address of the Zookeeper server to the constructor of the parent class. The parent class creates a !ZooKeeper instance if one does not exist. The constructor of Barrier then creates a barrier node on !ZooKeeper, which is the parent node of all process nodes, and we call root (obs: this is not the !ZooKeeper root "/").

```
      /**
******* Barrier constructor
*******
******* @param address
******* @param name
******* @param size
******* /
        Barrier(String address, String name, int size)
        throws KeeperException, InterruptedException, UnknownHostException {
            super(address);
            this.root = name;
            this.size = size;

            // Create barrier node
            if (zk != null) {
                    Stat s = zk.exists(root, false);
                    if (s == null) {
                        zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE, 0);
                    }
            }

            // My node name
            name = new String(InetAddress.getLocalHost().getCanonicalHostName().toString());
        }
```

To enter the barrier, a process calls enter(). The process creates a node under the root to represent it, using its host name to form the node name. It then wait until enough processes have entered the barrier. A process does it by checking the number of children the root node has with "getChildren()", and waiting for notifications in the case it does not have enough. To receive a notification when there is a change to the root node, a process has to set a watch, and does it through the call to "getChildren()". In the code, we have that "getChildren()" has two parameters. The first one states the node to read from, and the second is a boolean flag that enables the process to set a watch. In the code the flag is true.

```
    /**
******* Join barrier
*******
******* @return
******* @throws KeeperException
******* @throws InterruptedException
******* /

        boolean enter() throws KeeperException, InterruptedException{
            zk.create(root + "/" + name, new byte[0], Ids.OPEN_ACL_UNSAFE,
                    CreateFlags.EPHEMERAL);
            while (true) {
                synchronized (mutex) {
                    ArrayList<String> list = zk.getChildren(root, true);

                    if (list.size() < size) {
                        mutex.wait();
                    } else {
                        return true;
                    }
                }
            }
        }
```

Note that enter() throws both !KeeperException and !InterruptedException, so it is reponsibility of the application to catch and handle such exceptions.

Once the computation is finished, a process calls leave() to leave the barrier. First it deletes its corresponding node, and then it gets the children of the root node. If there is at least one child, then it waits for a notification (obs: note that the second parameter of the call to getChildren() is true, meaning that ! ZooKeeper has to set a watch on the the root node). Upon reception of a notification, it checks once more whether the root node has any child.

```
    /**
******* Wait until all reach barrier
*******
******* @return
******* @throws KeeperException
******* @throws InterruptedException
******* /

        boolean leave() throws KeeperException, InterruptedException{
            zk.delete(root + "/" + name, 0);
            while (true) {
                synchronized (mutex) {
                    ArrayList<String> list = zk.getChildren(root, true);
                        if (list.size() > 0) {
                            mutex.wait();
                        } else {
                            return true;
                        }
                    }
                }
       }
```

Producer-Consumer Queues

A producer-consumer queue is a distributed data structure that group of processes use to generate and consume items. Producer processes create new elements and add them to the queue. Consumer processes remove elements from the list, and process them. In this implementation, the elements are simple integers. The queue is represented by a root node, and to add an element to the queue, a producer process creates a new node, a child of the root node.

The following excerpt of code corresponds to the constructor of the object. As with Barrier objects, it first calls the constructor of the parent class, ! SyncPrimitive, that creates a !ZooKeeper object if one doesn't exist. It then verifies if the root node of the queue exists, and creates if it doesn't.

```
    /**
******* Constructor of producer-consumer queue
*******
******* @param address
******* @param name
******* /
        Queue(String address, String name)
        throws KeeperException, InterruptedException {
            super(address);
            this.root = name;
            // Create ZK node name
            if (zk != null) {
                    Stat s = zk.exists(root, false);
                    if (s == null) {
                        zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE, 0);
                    }
            }
       }
```

A producer process calls "produce()" to add an element to the queue, and passes an integer as an argument. To add an element to the queue, the method creates a new node using "create()", and uses the SEQUENCE flag to instruct !ZooKeeper to append the value of the sequencer counter associated to the root node. In this way, we impose a total order on the elements of the queue, thus guaranteeing that the oldest element of the queue is the next one consumed.

```
    /**
******* Add element to the queue.
*******
******* @param i
******* @return
******* /

        boolean produce(int i) throws KeeperException, InterruptedException{
            ByteBuffer b = ByteBuffer.allocate(4);
            byte[] value;

            // Add child with value i
            b.putInt(i);
            value = b.array();
            zk.create(root + "/element", value, Ids.OPEN_ACL_UNSAFE,
                        CreateFlags.SEQUENCE);

            return true;
        }
```

To consume an element, a consumer process obtains the children of the root node, reads the node with smallest counter value, and returns the element. Note that if there is a conflict, then one of the two contending processes won't be able to delete the node and the delete operation will throw an exception.

A call to getChildren() returns the list of children in lexicographic order. As lexicographic order does not necessary follow the numerical order of the counter values, we need to decide which element is the smallest. To decide which one has the smallest counter value, we traverse the list, and remove the prefix "element" from each one.

```
     /**
******* Remove first element from the queue.
*******
******* @return
******* @throws KeeperException
******* @throws InterruptedException
******* /
        int consume() throws KeeperException, InterruptedException{
            int retvalue = -1;
            Stat stat = null;

            // Get the first element available
            while (true) {
                synchronized (mutex) {
                    ArrayList<String> list = zk.getChildren(root, true);
                    if (list.isEmpty()) {
                        System.out.println("Going to wait");
                        mutex.wait();
                    } else {
                        Integer min = new Integer(list.get(0).substring(7));
                        for(String s : list){
                            Integer tempValue = new Integer(s.substring(7));
                            if(tempValue < min) min = tempValue;
                        }
                        System.out.println("Temporary value: " + root + "/element" + min);
                        byte[] b = zk.getData(root + "/element" + min, false, stat);
                        zk.delete(root + "/element" + min, 0);
                        ByteBuffer buffer = ByteBuffer.wrap(b);
                        retvalue = buffer.getInt();

                        return retvalue;
                    }
                }
            }
        }
```

You can also checkout the EurosysTutorial given at Eurosys 2011.