Introduction to Waterken Programming

Marc Stiegler, Jing Tie

HP Laboratories
HPL-2010-89

Keyword(s):

distributed secure reliable computing

Abstract:

Waterken is a platform built on the Java virtual machine that allows the quick development of reliable, secure, distributed applications. In this introduction, we will first go over the basic concepts of the Waterken platform. Then we will write two services. The first is a digital cash system. The second is a cookie store, at which you can spend digital cash and buy virtual cookies. After confirming that the store and the cash work together, we will initiate a cookie purchase while crashing the cash server and watch reliable recovery take place even though the application includes no recovery code.

External Posting Date: August 6, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: August 6, 2010 [Fulltext]

Introduction to Waterken Programming

by Marc Stiegler and Jing Tie

1 Overview

Waterken is a platform built on the Java virtual machine that allows the quick development of reliable, secure, distributed applications. In this introduction, we will first go over the basic concepts of the Waterken platform. Then we will write two services. The first is a digital cash system. The second is a cookie store, at which you can spend digital cash and buy virtual cookies. After confirming that the store and the cash work together, we will initiate a cookie purchase while crashing the cash server and watch reliable recovery take place even though the application includes no recovery code.

2 Concepts

1.1 Promise Pipelining

Waterken uses a nonlocking, nonblocking concurrency paradigm, *promise pipelining*, which is based on the actors model of concurrency. With promise pipelining on Waterken, programs are guaranteed to have neither deadlocks nor fine grain data races. This model conforms to the way people do distributed computation, as illustrated in the following story.

Alice, the CEO of Evernet Computing, needs a new version of the budget including R&D numbers from the VP of Engineering, Bob. Alice calls Bob: "Could you get me those numbers?" Bob jots Alice's request on his to-do list. "Sure, I promise I'll get them for you after I solve this engineering problem". Bob has handed Alice a promise for the answer. He has not handed her the answer. But neither Bob nor Alice sits on their hands, blocked, waiting for the resolution. Rather, Bob continues to work his current problem.

Meanwhile Alice goes to Carol, the CFO: "Carol, when Bob gets those numbers, plug them into the spreadsheet and give me the new budget, okay?" Carol: "No problem." Carol writes Alice's request on her own to-do list, but does not put it either first or last in the list. Rather, she puts it in the conditional part of the list, to be done when the condition is met--in this case, when Bob fulfills his promise.

Conceptually, Alice has handed to Carol a copy of Bob's promise for numbers, and Carol has handed to Alice a promise for a new integrated spreadsheet. Once again, no one waits around, blocked. Carol ambles down the hall for a contract negotiation, and Alice goes back to preparing for the IPO.

When Bob finishes his calculations, he signals that his promise has been fulfilled. When Carol receives the signal, she uses Bob's fulfilled promise to fulfill her own promise. When Carol fulfills her promise, Alice gets her spreadsheet. A sophisticated distributed computation has been completed so simply that no one realizes an advanced degree in computer science should have been required.

In this simple example, we see why human beings never get trapped in the thread-based deadlock situations described endlessly in books on concurrent programming. People don't deadlock because they live in a

concurrent world managed with a promise-based architecture.

The Waterken unit of concurrent computation is a *vat*, which hosts a collection of objects. In addition to holding a set of interrelated objects, the vat has a single thread of execution, and an event queue. Objects inside a vat can speak to each other in the usual OO fashion using *immediate calls*: object A sends a message to object B and waits for the answer, with execution on the single thread moving from A to B and back to A. Objects in other vats can only communicate with an object inside the vat using an *eventual send*, which places a message on the vat's event queue. The Waterken infrastructure takes one message off the queue at a time, runs the event initiated by that event to completion, and then moves on to the next message.

When an object in vat Alice sends a message to an object in vat Bob, the answer comes back only eventually. However, processing in vat Alice continues to the next statement immediately, because the Waterken platform creates a local placeholder for the answer that can be stored and used until the answer is eventually delivered. This placeholder may be either a *promise* or a *proxy*.

If the answer being returned is a primitive immutable object such as String, Integer, Float, or Boolean, the sender immediately receives a *promise* for the answer. This promise is an object that can be sent to other objects just like any other object. Of course, there are times when you need the concrete value, perhaps to branch on the value. In that case, you set up a special Waterken construct called a *when-block* that will put a block of code on the event queue when the promise is finally *resolved* with a *fulfilled* value.

If the answer being returned is a reference to a mutable object that lives in another vat, instead of getting a promise as in the above case, you immediately get a reference to a *proxy*. The proxy is an object that can be passed to other objects just like any other object. In addition, you can also immediately start sending messages to the object's host vat through the proxy. These messages will be delivered by the host vat to the actual object once the object has been computed. This means the message can be waiting locally for the actual object when it is identified, i.e. no round-trip is required to get a reference to the actual object before sending the message to it. Messages from one vat to another are delivered to the recipient’s queue in the order they were sent.

More formally, if an object is pass-by-reference (including all the objects belonging to new classes that you create in this tutorial), these objects live inside the vat where they are created, they never leave that vat, and they are represented by proxies in other vats. Primitive immutable types are passed by copy: if you hand someone a 32-bit integer, instead of handing them a reference to the integer, the 32 bits of the integer are simply copied and given to them. Such pass by copy objects, when returned from a method invocation on an object in another vat, are represented by promises. When the promise is fulfilled, the object's bits are duplicated in the client vat that asked for the value. We will see shortly how to set up a when-block to use the fulfilled copy of the data in a promise. While you can also use a when-block to wait for a proxy to be fulfilled, this is generally not necessary because the proxy is fully functional from the moment it is constructed: messages sent to the proxy are delivered to the actual object as soon as possible.

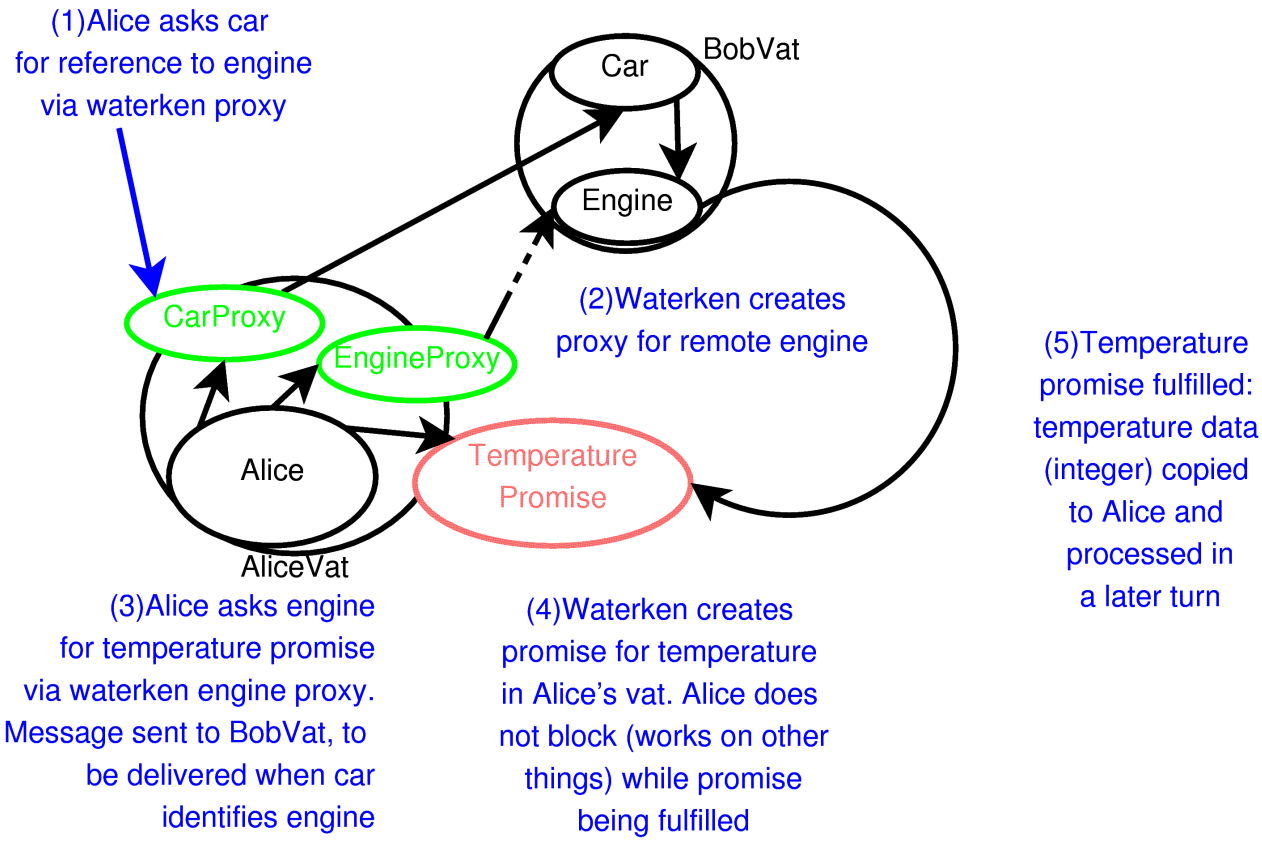
Let us look at how these parts interrelate with a one-line example: Alice remotely asks Bob’s car’s engine for its temperature. The following Java code in Alice would be valid either in a Waterken framework or when using Java RMI, though the computational consequences are quite different:

```
alice.testTemperature(bobCar.engine().getTemperature());
```

In Java RMI, the first step is to call the bobCar to get a reference to its engine. When the engine reference returns to the caller Alice, the caller then sends the next message to ask for the temperature. When the answer for the temperature returns to Alice, the temperature can then be used. This computation requires 2 round

trips before the computation can continue, once for the engine and once for the temperature.

In Waterken, the call to bobCar for the engine immediately produces a proxy for the engine on the caller side. (This is the engine proxy produced for Alice by Waterken) The request getTemperature is then immediately shipped through the engine proxy to the Bob's vat, to be delivered to the engine once the engine object is computed. Meanwhile, this transmission of getTemperature immediately generates a promise for the temperature on the sender's side. Alice sets up a when-block on the promise for the temperature, to assess the temperature when the promise is fulfilled, and then moves on to do other things. Alice does not block and wait for either the engine or the temperature. There is only one round trip, for the temperature.



The above sequence in Waterken is also shown in the figure:

1. Alice asks the car (via her CarProxy) for a reference to the engine via a Waterken CarProxy.

2. Waterken creates proxy for remote engine, the EngineProxy.

3. Alice asks the engine for the temperature promise via the new EngineProxy. The message is sent to BobVat, to be delivered when car identifies the engine.

4. Alice sends the temperature promise to the Analyzer.

5. The temperature promise is fulfilled: the temperature data (an integer) is copied to Carol.

2.2 Automatic Checkpointing

Another important feature of the Waterken platform is its transparent automatic checkpointing. The sequence of actions that starts with picking a message off the event queue, then delivering it to the recipient object, and processing the message until an answer is ready, is known as a *turn*. A vat runs one turn to completion, and then runs the next turn. If the state of objects is changed or a message is sent to a remote object during a turn, a checkpoint is taken automatically at the end of the turn as a part of the process of returning the answer and sending new eventual messages that were initiated during the turn itself. As a consequence, the programmer never has to worry about where a might vat crash and doesn't need to know whether a particular message was sent, or whether it was received. The platform automatically processes the turns in such a fashion that every vat's local checkpoint is always in a consistent state, and eventually all the messages will be processed and answered with valid results. A Waterken server that is shut down either intentionally or because of a system failure can simply be restarted from its most recent checkpoint, and computation will pick up where it left off. There is never a special shutdown sequence: one can always kill a vat by killing the java process in which it is running without fear of inconsistency or loss of data .

2.3 User Interface and Webkeys

User interfaces for Waterken servers are usually written in JavaScript stored on the server with the application. The urls generated by Waterken to reference objects in an application (called webkeys) are specially formulated to do the encryption, authentication, and authorization required for secure operation. In this first introduction, we will use JavaScript code and webkeys to view some of our objects, though discussion of how to create such JavaScript is beyond the scope of this document.

3 Waterken Installation

To compile and run the services and exercises below, you will first need to install a Waterken Server on your machine. The Java JDK 1.6 or later is required. Waterken can be found at

http://waterken.sourceforge.net/

The folder you have at the end of installation will be the launch and execution folder for a Waterken server. Follow the instructions on the Waterken site for using genkey to create a public/private key pair for this server. Remember, when running genkey, to specify the proxy settings if you are behind a firewall that requires a proxy.

4 Start Simple Services

4.1 Purse Service

Our first service will be a digital cash system based on the Purse protocol. The basic idea is that money is stored in a purse. To withdraw money from a purse, you create a new purse and put some of the money from the original purse into the second one. To deposit, you hand a second purse to the main purse; all the money

is then moved from the second purse to the main purse.

You can see the user interface to a purse system at the following webkey:

https://sha-256-hl6w2x74ixy6pi5n.yurl.net:4445/-/tutbucks/#s=ashzre7yp5wauo

In Waterken, the purse is represented by a Java interface with the following signature. This code assumes that you have created the package org.waterken.tutorial to hold the code for this tutorial.

```
package org.waterken.tutorial;
import org.ref_send.promise.Promise;
public interface Purse {
   Promise<Integer> getBalance();
   Purse withdraw(int amount);
   Promise<Integer> deposit(Purse purse);
}
```

As described earlier, when returning a concrete pass-by-copy value from one vat to another, you must return a promise. Hence getBalance() returns a promise for an integer, not an integer itself. The withdraw method returns a purse. Note that the returned purse is not wrapped in a promise because the purse is a pass by reference object. Consequently, a proxy for the purse will be constructed on the client's side. In Java, proxies can only be constructed for interface types, not concrete types, so the returned value must be an interface. The deposit method returns the amount that was deposited, i.e., the amount that was originally contained in the purse sent as an argument.

The general rules for the categories of objects that can be returned from public methods and that can be received as arguments by public methods are as follows:

- A pass by copy object returned from a method invocation across vats must be wrapped in a promise, such as Promise<Integer>.
- A pass by reference object returned from a method invocation across vats must be of an interface type, such as Purse.
- A pass by reference object sent as an argument to a method across vats must be of an interface type, such as Purse.
- A pass by copy object sent as an argument to a method across vats can be a raw object, such as an int (though you can pass promises, such as Promise<Integer>, to a method across vats as well). We see this above in withdraw(int).

Money systems have important security requirements. In the purse service on Waterken, the most important security property will be supplied by the fact that, due to the nature of webkeys, you cannot access a purse unless someone who already has access to the purse explicitly hands you a reference to it.

Here is an implementation of the purse.

```
package org.waterken.tutorial;
import java.io.Serializable;
import org.ref_send.promise.Eventual;
```

```
import org.ref_send.promise.Promise;
public class PurseX implements Purse, Serializable {
    private static final long serialVersionUID = 1L;
    private int balance;

    private PurseX(int initialBalance) {
        balance = initialBalance;
    }
    public static Purse make(Eventual q) {
        return new PurseX(100000);
    }

    public Promise<Integer> getBalance() {
        return Eventual.ref(balance);
    }
    public Purse withdraw(int amount) {
        if (amount < 0 || amount > balance) {
            throw new EventException("bad amount");
        }
        Purse newPurse = new PurseX(amount);
        balance -= amount;
        return newPurse;
    }
    public Promise<Integer> deposit(Purse purse) {
        PurseX realPurse = (PurseX)purse;
        int amount = realPurse.balance;
        realPurse.balance = 0;
        balance += amount;
        return Eventual.ref(amount);
    }
}
```

The PurseX class must implement the Purse interface so that proxies can be constructed for it. It must also implement the Serializable interface so that Waterken's checkpointing system can save the state of the purse at the end of a turn, thus guaranteeing the purse is always in a consistent state: it is either in the state from before the beginning of the turn (if there is a power failure or other catastrophe before the turn completes, it reverts to the original state), or at the end of the turn (when the checkpoint is taken, before the answer is returned).

For a class to be used to construct the root object of a new vat, the class must implement a public static make(Eventual) method that returns an interface type (in this case the PurseX make method returns an object of type Purse). The Eventual object passed into the make method is used to interact with the vat's event queue; we will see it in use when we build the cookie shop service. In the make method above, a new PurseX is constructed that contains 100000 credits. This purse becomes both the root of the vat and also the root of the currency that we have created, i.e., all the money that will ever exist in this currency is created and placed in this first purse during construction. Inflation is not an option.

The getBalance() method uses the Eventual.ref static method to wrap a promise around the balance instance variable and returns it. The Eventual class is part of the Waterken library, in the ref_send package. Many methods useful to the application programmer are found in the ref_send package, we will see more of them in the course of this tutorial.

The withdraw(amount) method creates a new purse with the specified amount, deducts that amount from the current Purse's balance, and returns the new purse. If an invalid amount was requested, an exception is

thrown back to the sender, and the proxy for the withdrawal purse winds up pointing to the exception rather than an actual purse. Further messages sent to this *broken* proxy will result in the creation of more broken proxies and broken promises: the brokenness propagates much like the IEEE floating point NaN value propagates. Broken promises and broken proxies can be caught and handled in the when-block, which will be described when we build the cookie shop.

The deposit(purse) method first casts the purse back into a concrete PurseX. This serves 2 purposes. The primary purpose is to give the current purse access to the depository purse's instance variables so that the depository purse's balance can be set to zero as part of moving the money to the recipient purse. The other purpose is to prevent fraud: if a malicious object were to create a fake purse (i.e., an object with a Purse interface but which is not a purse belonging to this currency), the cast of the purse to a PurseX would throw an exception, which results in a broken promise for the amount deposited. Once we have cast the argument to a PurseX, we move the money to the recipient purse and return the amount of the deposit, again wrapped in a promise.

To create the Purse service, create a project folder named "tutorial" in the root folder of your Waterken system; this tutorial folder should be a peer with other folders including ref_send and config. Configure your environment so that there are parallel bin and src hierarchies in the tutorial folder. Your Purse and PurseX code go into tutorial/src/org/waterken/tutorial (i.e., make sure they go into the org.waterken.tutorial package). Ensure that the .class files go into parallel locations in the bin hierarchy. Navigate into the org.waterken.tutorial package place and compile a java file named package-info.java with a single line:

@org.joe_e.IsJoeE package org.waterken.tutorial;

Every package you create needs a package-info.java file like this.

Next open a command line whose working directory is the Waterken root directory (a listing of the directory should include your tutorial folder). Now spawn a vat with a Purse as the root object by running the command

java -jar spawn.jar tutorial org.waterken.tutorial.PurseX cashserver

which follows the template

java -jar spawn.jar projectfoldername rootobjectclasspath vatfoldername

If this operation fails, there are a number of things that may have gone wrong. Review this checklist, a compilation of errors that have been made by others spawning their first vat:

- Make sure you included the package-info.java file in org.waterken.tutorial, and make sure you compiled it into a binary that shows up along with the other .class files for the package
- Make sure you ran genkey to create a public/private key pair for the server. You should see config/keys.jks, config/registration.json, and config/updateDNS.json. To re-run genkey you will need to delete those files. If you are behind a firewall that requires using a proxy, set the proxy specification as part of the genkey command line, such as
    - -DproxyHost=web-proxy.hpl.hp.com -DproxyPort=8088 -DproxySet=true
- Make sure that the ports specified for http and https, found in config/http.json and config/https.json, do not conflict with another service on your machine.
- To try again to create the vat, you must make sure that you do not have the folder config/vat/cashserver, or the file config/vat/.cashserver.was

- Ensure that the Purse interface is public
- Ensure that the PurseX class is public
- Ensure that the PurseX *make* method is both public and static.
- Ensure that the full classpath is included for the rootobjectclasspath on the spawn command.
- Ensure that there are no accidental spaces in the arguments on the spawn command. It is particularly easy to accidentally embed a space in the rootobjectclasspath.

When the spawn command executes successfully, it finds the PurseX class in the tutorial project and uses PurseX's make(Eventual) method to create a new Purse object in a new vat. The new vat has a checkpointing folder, config/vat/cashserver. Once the vat has been created, the command will return a webkey that is a reference to the newly created purse object. Save this webkey! The most convenient way to save it is as a bookmark in a web browser.

Now launch the server by using the following command:

java -jar serve.jar

If you are behind a firewall you may need to specify the proxy parameters for the JVM as part of the launch.For example,

java -DproxyHost=web-proxy.hpl.hp.com -DproxyPort=8088 -DproxySet=true -jar serve.jar

launches a server behind the HP firewall. The server will come to life and output a few messages about its launch.

If you are running the Vista operating system, your firewall will try very hard to prevent this launch from succeeding. You must not only turn off the firewall's blocking of the server port temporarily, you must go into the firewall software's configuration system and permanently unblock the server's port. If you do not shut off the firewall's block permanently, it will silently re-enable itself shortly thereafter and leave you puzzled as to why no messages are getting to the server.

If the serve command fails, run through the checklist described earlier for possible reasons for the spawn command to fail.

You can go to the webkey location using your browser. The Waterken server is pre-configured with user interfaces for both the Purse and the CookieShop. When you go to the Purse’s page using the webkey, you should see the Purse’s user interface. This page tells you the balance of your root purse and offers fields for making a withdrawal or making a deposit. Type a quantity (like "25") into the withdrawal field and press the button. A new link will appear on the page that is actually the webkey for the new purse. Note that the balance for your root purse has been reduced; the deduction matches the amount in the new purse. Open a new window on the new purse; it will have the same fields. Copy the webkey of the new purse from the location bar and paste it into the deposit field on the original root purse, and click deposit. The balance of the root purse will return to its original value; if you refresh the window on the new purse, you will find it now has a zero balance.

4.2 Cookie Shop Service

You can buy cookies from the cookie shop using the money from your purse system; the cookies will taste every bit as good as the credits in your purse. The interface to the cookie shop has the following methods:

```
public Promise<Integer> sellCookies(Purse payment);
```

```
    public Promise<Integer> getCookiesSold();
    public Promise<Integer> getRevenues();
```

To buy cookies with the sellCookies method, you must give the cookie shop a purse for payment. You will receive one cookie for every 3 credits in the purse; the method returns a count of the number of cookies you have bought. The getCookiesSold method tells how many cookies have been sold in total, and the getRevenues method tells how much money has been collected.

You can see an example CookieShop at

https://sha-256-hl6w2x74ixy6pi5n.yurl.net:4445/-/cookieshop/#s=gijlimpujtfcho

This cookie shop works with the example Purse whose link was presented at the beginning of the section on the Purse Service. Create a Purse from that purse service with at least 3 credits, give it to the cookie shop, and you should see the number of cookies bought go up.

The source for both the interface and the concrete CookieShopX class are shown below. Note how the concrete class is included in the same file with the interface by making it a nested class. With Waterken, there are many interfaces for which there is only one concrete implementation. Combining the interface and the class in a single file avoids file proliferation in your development environment, at the cost of making the classpath to the concrete class more complicated, as we will see later.

```
package org.waterken.tutorial;
import java.io.Serializable;
import org.ref_send.promise.Eventual;
import org.ref_send.promise.Promise;
import org.ref_send.promise.Do;

public interface CookieShop {

    public Promise<Integer> sellCookies(Purse payment);
    public Promise<Integer> getCookiesSold();
    public Promise<Integer> getRevenues();

    public class CookieShopX implements CookieShop, Serializable {
        private static final long serialVersionUID = 1L;

        private Eventual q;
        private Purse revenues;
        private int totalCookiesSold = 0;

        private CookieShopX(Eventual q, Purse revenues ) {
            this.q = q;
            this.revenues = revenues;
        }

        public static CookieShop make(Eventual q, Purse purse) {
            return new CookieShopX(q, purse);
        }

        public Promise<Integer> getCookiesSold() {
            return Eventual.ref(totalCookiesSold);
        }

        public Promise<Integer> getRevenues() {
```

```
            return revenues.getBalance();
        }

        public Promise<Integer> sellCookies(Purse payment) {
            Promise<Integer> numSold = q.when(revenues.deposit(payment),
                new HandleCookieSale());
            return numSold;
        }

        class HandleCookieSale extends Do<Integer, Promise<Integer>>
            implements Serializable {
            private static final long serialVersionUID = 1L;
            @Override public Promise<Integer>
            fulfill(Integer amountPaid) throws Exception {
                int freshCookiesSold = amountPaid/3;
                totalCookiesSold += freshCookiesSold;
                return Eventual.ref(freshCookiesSold);
            }
        }
    }
}
```

The first interesting feature is the extra argument we are requesting in the make(q, purse) method. It is possible to give the root object of a vat a set of references (presented as webkeys on the spawn.jar command line when you create the vat) at time of construction. Here the cookie shop is endowed at birth with a reference to a purse that it will use to contain its revenues.

The most important new machinery found in the CookieShop is in the sellCookies method, which makes a transaction using the purse service. SellCookies contains a when-block, built using the *when* method on the Eventual q object. A q.when expects 2 arguments: a promise that will be eventually resolved, and an object from a subclass of Do that will be fired when that resolution takes place. Usually, firing the when-block involves calling the Do object's fulfill method, but if the promise is broken it will call the Do object's reject method (which has a default implementation in the Do superclass).

The Do object and the q.when return an object (typically a promise, or a Void) that will be resolved when the when-block fires. In this example, q.when return a Promise<Integer> which is the number of cookies just sold. It gets this value from a freshly constructed object from the nested class HandleCookieSale, a subclass of Do.

In this example, the deposit(payment) method returns a promise for an integer that is the amount of money moved from the payment purse to the revenues purse. Therefore, when the payment is deposited and the size of the payment resolves, the *fulfill* method in the HandleCookieSale object is fired with the deposit amount as the argument. The number of cookies sold in this transaction is computed, the total number of cookies sold is updated, and the promise for the number of cookies sold is fulfilled.

4.3 Fault Tolerance

Now let’s initiate a cookie purchase while crashing the cash server and watch recovery take place even though no code was written to survive failures. First create a purse service. It is possible to run many services on a single Waterken server. You could use spawn.jar to create a new vat with the cookie shop on the same server as the purse. However, as part of this tutorial, we are interested in studying Waterken's reliability-despite-failure nature, and so we will put the shop on a separate server so that we can independently shut each server off and on, simulating various failures.

Make a copy of the entire Waterken folder. In the new folder, delete the files config/keys.jks, config/registration.json, and config/updateDNS.json: these are files associated with the public/private key pair that uniquely identifies this server, and are created by running genkey.jar. Having deleted these files, run genkey.jar for this server as you ran it earlier for the first server. Now open https.json and http.json with a text editor, and change the port numbers -- if we left the port numbers the same, and tried to run this server on the same computer with the first server, the second launch would fail because the port number was already taken, producing a bind error. Go into config/vat and delete the cashserver folder and the ".cashserver.was" file to eliminate this (invalid in the new server) copy of our purse vat. Open a browser on your purse, and create a new purse with no money in it by withdrawing an amount of 0. Save that webkey: this is the purse that you will give to the cookie shop at creation time so that it can deposit payments.

At this point the second server is configured, and we can create our cookie shop vat. With the root folder of the second server as a working directory, as described earlier, on a shell command line enter

java -jar spawn.jar tutorial org.waterken.tutorial.CookieShop$CookieShopX cookieshop *@pursewebkey*

while replacing the phrase “pursewebkey” with the actual webkey for the purse. Note the "@" sign preceding the pursewebkey, which tells the spawn application to convert this argument into a reference: if there is no "@" sign, the argument is passed to the application as a simple string. If you are using Linux with a bash command shell, you will need to put a backslash in front of the dollar sign, i.e., "\$".

When the spawn succeeds, capture the resulting webkey as a bookmark.

Now launch the cookie shop server (using serve.jar) and point a browser at the cookie shop webkey we got from the spawn operation. You should see a place where the webkey for a purse can be placed. Create another purse with some money (perhaps 9 credits) from your main purse, and use its webkey to buy cookies. You should see the number of cookies updated in the cookieshop window. You can use the revenues purse webkey that you gave to the cookie shop on the cookie shop spawn command line to see the revenues in the purse directly.

So far so good. But what if something goes wrong? What if one of the services in your distributed system crashes, and is unavailable for a while? Create another purse with money to purchase cookies. Now kill the purse server, and buy cookies from the shop. The shop cannot complete the transaction, of course: it can't deposit the purse without the purse server running, so the promise for the amount deposited will not be resolved, and the fulfill method on the HandleCookiesSold object will not be fired. Now turn the purse server back on. Since the purse server was not available when the Waterken server underneath the shop first tried to reach it, the shop server continued to try periodically to reach it. When the purse server comes back up, eventually the cookie server tries again and succeeds. Then the JavaScript in your browser (which is using the same promise pipelining techniques as the Java in the Waterken platform) eventually polls for the answer for the number of cookies just sold, and refreshes the screen. Since both your browser and your shop server are polling only periodically, it may take a while for the transaction to be complete and your screen to be refreshed, but eventually the right thing happens: you have here an unusually reliable distributed system. And you didn't have to write any special code to make it happen.

These servers and their services can also move around to different machines on different IP addresses and even different domain names, and the other services that use them will still find them. When a Waterken server launches, it registers its IP address with a redirectory service. If an object in a vat on another server tries to send a message to an object on the moved server, the Waterken infrastructure will ask the redirectory where it is and route traffic to the new location (this will fail if the sender of the message and the recipient of the message wind up separated by an appropriately hostile firewall). Hence you can create an interdependent

set of servers on one machine, test them as a networked application, and then move them to their production locations and have them continue operating without modification.

5 Conclusion

In this introduction, we have created multiple Waterken servers with separate vats, and connected the objects in those vats into a reliable and secure distributed system. We have created promises and fulfilled them. Additional references are found in the next section.

6 References

Waterken web site, http://waterken.sourceforge.net/, Tyler Close.

"E in a Walnut", http://www.skyhunter.com/marcs/ewalnut.html#SEC18, Marc Stiegler, excerpts used with permission of the author.

"Towards Fearless Distributed Computing", Marc Stiegler, HP Tech Report HPL-2009-258, http://www.hpl.hp.com/techreports/2009/HPL-2009-258.html