

A Programmer's Tutorial on Event-Driven Programming, Asynchronous Input/Output, and the Bamboo DHT

Sean C. Rhea
srhea@cs.berkeley.edu

December 15, 2005

1 Preliminaries

This document is a tutorial on three related topics: event-driven programming, asynchronous input/output (I/O), and the Bamboo DHT. I assume you're already familiar with Java, including the new language features for generics (called templates in C++) introduced in Java 1.5. If you're not yet aware of what generics are or how they work, please first read "Using and Programming Generics in J2SE 5.0" on Sun's web page.¹ Most of the example code shown in this tutorial is available on the Bamboo website.² I encourage you to download, run, modify, and re-run the example programs; there's no better way to learn a new programming style or library.

2 Introduction

We'll start the tutorial with an example: assume that you are writing a high-performance web server. A main loop for the web server might look something like this:

```
while (true) {
    Socket sock = acceptConnection();
    HttpRequest req = parseRequest(sock);
    HttpResponse resp = createResponse(req);
    sendResponse(sock, resp);
    sock.close();
}
```

Presumably, this web server will have a high-bandwidth connection to the Internet. On the other hand, many of the clients of the web server may have slow connections, meaning that your code will spend a lot of time in the `sendResponse` function. In effect, the performance of your web server is limited by the bandwidth of whatever client it's serving at any given time. (That's bad.)

To get around this performance limitation, we need to introduce *concurrency*: the ability to serve more than one client at a time. Historically, there have been at least two popular approaches to adding concurrency to a program. One approach is called *multithreading*. In this approach, the above code would be rewritten like this:

¹<http://java.sun.com/developer/technicalArticles/J2SE/generics/>
²WHAT

```

while (true) {
    final Socket sock = acceptConnection();
    Thread t = new Thread() {
        public void run() {
            HttpRequest req = parseRequest(sock);
            HttpResponse resp = createResponse(req);
            sendResponse(sock, resp);
            sock.close();
        }
    };
    t.start();
}

```

This code is much better. Since we have a separate thread for each request, and since each thread will run independently of all the others, it won't matter if one client has a very slow connection anymore; the `sendResponse` function will still take a long time to complete for that client, but it will run in parallel with the `sendResponse` functions for all the other clients we're servicing at the same time, rather than blocking them as before.

The above code is still not perfect, however. First of all, creating a new thread is an expensive operation in Java, and programs written like the one above can often spend much of their time creating new threads. While this problem can be solved with a construct called a *thread pool*, there is yet another problem with this style of code: most modern operating systems do not perform well when the number of Java threads running on them passes beyond several hundreds. Why this performance problem occurs is not important here, but it does mean that we can't expect a web server written in the above style to serve more than a couple of hundred clients at a time.

To work around this performance problem, we need to change our style of programming to what is often called *event-driven* style.³ The main idea of this style of programming is that given any function call that might block on input or output, such as the `sendResponse` function above, we split that function into a *request event* and a *response event*, normally represented as function objects. For example, we might define new versions of `parseRequest` and `sendResponse` like this:

```

public interface ParseDone {
    public void run(HttpRequest req);
}
void parseRequest(Socket sock, ParseDone done) { ... }
void sendResponse(Socket sock, HttpResponse resp, Runnable done) { ... }

```

The idea is that `parseRequest` will now return immediately after being called, and sometime later, when the request has actually been read off the network and parsed, the `run` function of the `ParseDone` object passed into `parseRequest` will be called. Likewise, `sendResponse` will also return immediately, and the `run` function of the `Runnable` object passed into `sendResponse` will be called later, once the response has been sent.

Ignore for a moment how these new versions of `parseRequest` and `sendResponse` are implemented, and consider only how we might use them. For example, we could rewrite the web server's main loop like this:

```

1 while (true) {
2     final Socket sock = acceptConnection();

```

³If you've ever programmed using Java's Swing graphics library, you've already done some event-driven programming.

```

3     ParseDone parseDone = new ParseDone() {
4         public void run(HttpRequest req) {
5             HttpResponse resp = createResponse(req);
6             Runnable sendResponseDone = new Runnable() {
7                 public void run() { sock.close(); }
8             };
9             sendResponse(sock, resp, sendResponseDone);
10        }
11    };
12    parseRequest(sock, parseDone);
13 }

```

Wow, that's some ugly code! But look past its ugliness for the moment, and let's think about what it does. First, on line 2 we accept a connection as before. Then, we decide what we're going to do once we've parsed a request and write it as a `ParseDone` object on lines 3–11. Next, we call `parseRequest` on line 12. That will parse the request, and then call `run` on line 4. In turn, `run` creates a response (line 5) and decides what it's going to do once the response is sent, encoding that knowledge as an object of type `Runnable`. It then calls `sendResponse`, which sends the response and then calls `run` on line 7, which closes the socket. Whew!

You're probably thinking that the above code is one seriously convoluted way to write a web server. In part you're right, although we'll improve on the readability of the above code later in this tutorial. At the same time, however, notice that we've both eliminated the problem of clients blocking on one another and threads from the program. Since there are no longer any blocking calls, each run of the while loop will now happen as quickly as possible, bringing us right back to the call to `acceptConnection` where we can begin helping the next client right away. And if you trust me for the moment that we can implement the new versions of `parseRequest` and `sendResponse` with only a single thread, then we've also eliminated our previous need for multiple threads.

The remainder of this document is in several parts. First, I'll describe the event model that Bamboo uses, and the various Bamboo classes that make event-driven programming easier. I won't describe how these classes are implemented in this tutorial, for that you'll have to read the source if you're interested. (It's not necessary to understand the source in order to use it.) Next, I'll describe how to use the Bamboo messaging layer and DHT router, so that you can build your own DHT applications on top of Bamboo.

3 The Bamboo Event Model

In this section I'll get into the full details of the event model Bamboo uses, starting from the beginning.

3.1 Callbacks and `registerTimer`

Above we saw a use of the interface `java.lang.Runnable`, which is defined like this:

```
public interface Runnable { public void run(); }
```

This kind of interface is often called a *callback*; you pass an object of type `Runnable` to some other function so that it can “call you back” later by calling the `run` function. Above, we passed one such object as an argument to the `sendResponse` function.

Events in Bamboo are implemented as either functions or callbacks. This idea is perhaps best explained by example. Bamboo includes a class, `bamboo.lss.AsyncCore` that contains two functions, `registerTimer` and `asyncMain`. You use them like this:

```

1 public class DelayedHelloWorld {
2     public static Runnable printCallback = new Runnable() {
3         public void run() {
4             System.out.println("Hello, world!");
5             System.exit(0);
6         }
7     };
8     public static void main(String [] args) throws java.io.IOException {
9         bamboo.lss.ASyncCore acore = new bamboo.lss.ASyncCoreImpl();
10        acore.registerTimer(5000, printCallback);
11        acore.asyncMain();
12    }
13 }

```

When you run this program, it will pause for five seconds and then print “Hello, World!” before exiting. Let’s break down the code a line at a time.

First, lines 2–7 define a callback, `printCallback`, that prints “Hello, world.” to standard output when it’s called. Line 9 creates an instance of `ASyncCore`, and line 10 tells that object to call `printCallback` in 5 seconds (5,000 milliseconds). Finally, line 11 calls `asyncMain`; once this function is called, `ASyncCore` is in charge. The function `asyncMain` never exits, it simply loops forever, calling the callbacks registered with `registerTimer` when required. (That’s why we call `exit` on line 5; otherwise, the program would run forever. Remove that call and try it to see what I mean.)

In your event-driven programs, you will usually create an object of type `ASyncCore`, perform some other setup including registering timer events, and then call `asyncMain`.

3.2 Passing Arguments to Callbacks

Let’s say that instead of wanting to print “Hello, World!” after five seconds, you instead wanted to print whatever string was passed in on the command line. You might try something like this:

```

1 public class BadDelayedEcho {
2     public static String str;
3     public static Runnable printCallback = new Runnable() {
4         public void run() {
5             System.out.println(str);
6             System.exit(0);
7         }
8     };
9     public static void main(String [] args) throws java.io.IOException {
10        if (args.length < 1) {
11            System.err.println("usage: java BadDelayedEcho <string>");
12            System.exit(1);
13        }
14        str = args[0];
15        bamboo.lss.ASyncCore acore = new bamboo.lss.ASyncCoreImpl();
16        acore.registerTimer(5000, printCallback);
17        acore.asyncMain();
18    }
19 }

```

Here we’ve used what amounts to a global variable, `str`, to pass the string to echo from the `main` function to `printCallback`. Bad programmer! Maybe you’re crafty, though, and you’ve already thought of a way around using a global variable, such as this:

```

1 public class BetterDelayedEcho {
2     public static void main(String [] args) throws java.io.IOException {
3         if (args.length < 1) {
4             System.err.println("usage: java BetterDelayedEcho <string>");
5             System.exit(1);
6         }
7         bamboo.lss.ASyncCore acore = new bamboo.lss.ASyncCoreImpl();
8         final String str = args[0];
9         Runnable printCallback = new Runnable() {
10            public void run() {
11                System.out.println(str);
12                System.exit(0);
13            }
14        };
15        acore.registerTimer(5000, printCallback);
16        acore.asyncMain();
17    }
18 }

```

In this program, we've removed the global variable by defining `printCallback` inside of `main`, and thereby putting `str` within its scope. Pretty crafty, but it has one big downside: what if `printCallback` itself defined another callback, which in turn defined another callback, and so on? Pretty soon you'd suffer death through excessive indentation.

3.3 The curry Function

To pass arguments to callbacks without using so much indentation, we'll introduce a technique from *functional* programming languages (such as Lisp and ML) called *currying*. In such languages, given a function that takes two arguments and a value for one argument, you can produce a new function that takes only the second argument. For example, using Java-like syntax, we might have:

```

public int add(int left, int right) { return left + right; }
public int add42(int right) = curry(add, 42);
...
int j = add42(7);

```

In other words, we curry the function `add` with the value `42` to produce a new function, `add42`, that takes one argument and returns the value of that argument plus `42`. Right now this technique might seem a little pointless, but bear with me; it will be a useful technique for passing arguments to callbacks.

It turns out that we can use Java's support for generics (called templates in C++) to implement currying. All we need is a way to refer to functions as objects, and we'll use callbacks for this purpose. Because of this decision, we will sometimes refer to callbacks as *function objects*.

We already have a callback that takes no arguments: the `Runnable` interface used above. What we'll do is introduce another type of callback that takes one argument like this:

```

public interface Think1<T> { void run(T t); }

```

Think is the term used in functional programming for a function that has return type `void`. We've called the above interface `Think1` because it takes one argument and returns `void`. We can create a new `Think1` object that takes an integer argument like this:

```

Thunk1<Integer> intThunk = new Thunk1<Integer>() {
    void run(Integer i) { System.out.println(i); }
}

```

Moreover, we can create a curry function that will take a Thunk1<T> and an object of type T and return a Runnable object like this:

```

public static <T> Runnable curry(final Thunk1<T> f, final T t) {
    return new Runnable() { public void run() { f.run(t); } };
}

```

We can then use this function to create a function that prints 42 like this:

```

Runnable print42 = curry(intThunk, new Integer(42));
print42.run(); // prints "42" to standard output

```

Bamboo contains a class, `bamboo.util.Curry`, that has definitions for Thunk1 through Thunk9, and the appropriate curry functions for all combinations of them. You can, for example, call `curry` with a Thunk7 and two arguments to get a Thunk5. To use all of these thunks and curry functions in your code, include this line at the top of your source file:

```

import static bamboo.util.Curry.*;

```

Now, if you remember, we were trying to improve the `BetterDelayedEcho` program above. Here's a new version using thunks and curries:

```

1 import static bamboo.util.Curry.*;
2 public class DelayedEcho {
3     public static Thunk1<String> printCallback = new Thunk1<String>() {
4         public void run(String str) {
5             System.out.println(str);
6             System.exit(0);
7         }
8     };
9     public static void main(String [] args) throws java.io.IOException {
10        if (args.length < 1) {
11            System.err.println("usage: java DelayedEcho <string>");
12            System.exit(1);
13        }
14        bamboo.lss.ASyncCore acore = new bamboo.lss.ASyncCoreImpl();
15        acore.registerTimer(5000, curry(printCallback, args[0]));
16        acore.asyncMain();
17    }
18 }

```

Lines 3-8 of this program create a new callback, this time one that takes a string argument and prints it to standard output. Unfortunately, the `registerTimer` function only takes arguments of type `Runnable`, so we can't pass it our new version of `printCallback` directly. We can, however, call `curry` with `printCallback` and the string we want passed to it as arguments to get an object of type `Runnable`, and we can then pass that to `registerTimer`. In other words, line 15 says, "after five seconds, call `printCallback` with the value in `args[0]`." Pretty cool, huh?

3.4 Using Channels

Timers are only one of the two main classes of callbacks that `ASyncCore` manages. The other class is for sockets, Unix's stream-based abstraction of the network. In Java 1.4 and later, sockets are accessed through objects called *channels*. There are `SocketChannels` for TCP clients, `ServerSocketChannels` for TCP servers, and `DatagramChannels` for UDP sockets.

The class `java.nio.channels.SelectionKey` defines the conditions you can wait on with a channel: `OP_ACCEPT`, `OP_CONNECT`, `OP_READ`, and `OP_WRITE`. To wait on any one of these conditions, call `registerSelectable` in `ASyncCore`; to stop waiting on them, call `unregisterSelectable`. Here's an example:

```
1 import bamboo.lss.*;
2 import java.io.IOException;
3 import java.net.*;
4 import java.nio.ByteBuffer;
5 import java.nio.channels.*;
6 import static bamboo.util.Curry.*;
7 import static java.nio.channels.SelectionKey.*;
8
9 public class HttpGet {
10
11     public static ASyncCore acore;
12
13     public static Thunk1<SocketChannel> acceptCallback =
14         new Thunk1<SocketChannel>() {
15         public void run(SocketChannel channel) {
16             try {
17                 if (channel.finishConnect()) {
18                     acore.unregisterSelectable(channel, OP_CONNECT);
19                     byte [] bytes = ("GET_/_HTTP/1.0\r\n\r\n").getBytes();
20                     ByteBuffer buf = ByteBuffer.wrap(bytes);
21                     acore.registerSelectable(channel, OP_WRITE,
22                         curry(writeCallback, channel, buf));
23                 }
24             }
25             catch (IOException e) {
26                 System.err.println("Could not connect: " + e);
27                 System.exit(1);
28             }
29         }
30     };
31
32     public static Thunk2<SocketChannel, ByteBuffer> writeCallback =
33         new Thunk2<SocketChannel, ByteBuffer>() {
34         public void run(SocketChannel channel, ByteBuffer buf) {
35             try {
36                 while (buf.position() < buf.limit()) {
37                     if (channel.write(buf) == 0)
38                         break;
39                 }
40                 if (buf.position() == buf.limit()) {
41                     acore.unregisterSelectable(channel, OP_WRITE);
42                     acore.registerSelectable(channel, OP_READ,
```

```

43         curry(readCallback , channel));
44     }
45 }
46 catch (IOException e) {
47     System.err.println("Could not write: " + e);
48     System.exit(1);
49 }
50 }
51 };
52
53 public static Thunk1<SocketChannel> readCallback =
54     new Thunk1<SocketChannel>() {
55     public void run(SocketChannel channel) {
56         while (true) {
57             try {
58                 ByteBuffer buf = ByteBuffer.wrap(new byte [1024]);
59                 int n = channel.read(buf);
60                 if (n > 0)
61                     System.out.print(new String(buf.array(), 0, n));
62                 else if (n == 0)
63                     break;
64                 else {
65                     channel.close();
66                     System.exit(0);
67                 }
68             }
69             catch (IOException e) {
70                 System.err.println("Could not read: " + e);
71                 System.exit(1);
72             }
73         }
74     }
75 };
76
77 public static void main(String [] args) throws IOException {
78     if (args.length < 1) {
79         System.err.println("usage: java HttpGet <host> <port>");
80         System.exit(1);
81     }
82     Acore = new ASyncCoreImpl();
83     SocketChannel channel = SocketChannel.open();
84     channel.configureBlocking(false);
85     channel.connect(new InetSocketAddress(
86         args[0], Integer.parseInt(args[1])));
87     acore.registerSelectable(channel, OP_CONNECT,
88         curry(acceptCallback , channel));
89     acore.asyncMain();
90 }
91 }

```

This program takes the name and port number of an HTTP server on the command line, connects to it using TCP, and sends the string "GET / HTTP/1.0" followed by two carriage-return, line-feed pairs. This string will cause any compliant HTTP server to return the resource named by "/", usually the file

index.html.

Important things to note include: on line 84 we call `configureBlocking(false)` on the `SocketChannel` object. If this is not done, calls to read or write the socket may block. Control will next move to the `acceptCallback` function once the TCP connection is established. After connecting on line 17, we unregister the accept callback on line 18. We then create a `ByteBuffer` object to hold the string we want to write, and register the the write callback function.

The function `SocketChannel.write` returns zero if no data can be written at the moment, and throws an `IOException` if the socket is closed. Once we have written all the data in the buffer, we unregister the write callback and register the read callback (lines 40–44).

Unlike `write`, `SocketChannel.read` has two completion cases. Like `write`, if the connection times out or some other error occurs, `read` will throw an `IOException`. Unlike `write`, if the remote node closes the connection normally, `read` will return -1. We read until either of these conditions occurs, echoing all bytes read to standard output.

```
1 import bamboo.lss.*;
2 import java.io.IOException;
3 import java.net.*;
4 import java.nio.ByteBuffer;
5 import java.nio.channels.*;
6 import java.util.LinkedList;
7 import static bamboo.util.Curry.*;
8 import static java.nio.channels.SelectionKey.*;
9
10 public class EchoServer {
11
12     public static ASyncCore acore;
13
14     public static Thunk1<ServerSocketChannel> acceptCallback =
15         new Thunk1<ServerSocketChannel>() {
16         public void run(ServerSocketChannel serverChannel) {
17             try {
18                 SocketChannel channel = serverChannel.accept();
19                 if (channel != null) {
20                     channel.configureBlocking(false);
21                     LinkedList<ByteBuffer> bufs =
22                         new LinkedList<ByteBuffer>();
23                     acore.registerSelectable(channel, OP_READ,
24                         curry(readCallback, channel, bufs));
25                 }
26             }
27             catch (IOException e) {
28                 System.err.println("Could not accept: " + e);
29                 System.exit(1);
30             }
31         }
32     };
33
34     public static Thunk2<SocketChannel, LinkedList<ByteBuffer>> readCallback =
35         new Thunk2<SocketChannel, LinkedList<ByteBuffer>>() {
36         public void run(SocketChannel channel, LinkedList<ByteBuffer> bufs) {
37             while (true) {
38                 ByteBuffer buf = ByteBuffer.wrap(new byte[1024]);
39                 try {
```

```

40         int n = channel.read(buf);
41         if (n > 0) {
42             buf.flip();
43             bufs.addLast(buf);
44             acore.registerSelectable(channel, OP_WRITE,
45                                     curry(writeCallback, channel, bufs));
46         }
47         else {
48             if (n < 0)
49                 throw new IOException("channel_closed");
50             break;
51         }
52     }
53     catch (IOException e1) {
54         try {
55             channel.close();
56             acore.unregisterSelectable(channel,
57                                       OP_READ | OP_WRITE);
58         }
59         catch (IOException e2) { /* Do nothing. */ }
60     }
61 }
62 }
63 };
64
65 public static Thunk2<SocketChannel, LinkedList<ByteBuffer>> writeCallback =
66     new Thunk2<SocketChannel, LinkedList<ByteBuffer>>() {
67     public void run(SocketChannel channel, LinkedList<ByteBuffer> bufs) {
68         try {
69             while (! bufs.isEmpty()) {
70                 ByteBuffer buf = bufs.getFirst();
71                 while (buf.position() < buf.limit()) {
72                     if (channel.write(buf) == 0)
73                         return;
74                 }
75                 if (buf.position() == buf.limit())
76                     bufs.removeFirst();
77             }
78             acore.unregisterSelectable(channel, OP_WRITE);
79         }
80         catch (IOException e) {
81             System.err.println("Could_not_write:_" + e);
82             System.exit(1);
83         }
84     }
85 };
86
87 public static void main(String [] args) throws IOException {
88     if (args.length < 1) {
89         System.err.println("usage: _java _EchoServer _<port>");
90         System.exit(1);
91     }
92     int port = Integer.parseInt(args[0]);
93     acore = new ASyncCoreImpl();

```

```
94     ServerSocketChannel channel = ServerSocketChannel.open();
95     channel.socket().bind(new InetSocketAddress(port));
96     channel.configureBlocking(false);
97     acore.registerSelectable(channel, OP_ACCEPT,
98         curry(acceptCallback, channel));
99     acore.asyncMain();
100 }
101 }
```