

BAMBOO - A Tutorial

Marcel Dischinger
(*marcel.dischinger@tm.uka.de*)

April 2004

Abstract

The goal of this tutorial is to show you how to build a simple BAMBOO application that can send simple text messages from one node to another. In addition it will give you a rough insight to the bamboo communication and event system. We assume that the reader has some knowledge of P2P systems (e.g. [?]) before starting. It cannot be our task to explain how a P2P system like BAMBOO works. Refer to the available documentation for BAMBOO [?] and Pastry [?] to learn more about how the BAMBOO router works and how the P2P system is organized. You also need to know how to write a programme in Java.

1 Introduction

When I started working with BAMBOO there was no tutorial. There are some examples of modules in the source tree but they are too complex and sometimes too inconsistent to learn from (beside a lack of comments and documentation for the simple structures used at each stage).

Bamboo is written in an event-driven, single-threaded programming style. In first place it inherits its structure from SEDA, which stands for the *Staged, Event-Driven Architecture*. As the name suggests, each part of a BAMBOO application is a *stage*; communication is done by passing events to each stage, while the whole application remains single threaded and serialized.

This tutorial is meant to be a short introduction to BAMBOO without giving a deep insight of the underlying system. It might prepare you to start writing your own stages using BAMBOO and to use the basic functions provided by the system to create your own stage.

This can only be a start. If you are going to write more complex stages, you will end up reading the source code provided, but by that time it will be easier because you know the basics. This Tutorial does not deal with the BAMBOO DHT API.

BAMBOO uses a lot of code from the OceanStore project. You can get the source code and Java-Doc from the tapestry project [?], if you would like to take a look at it. You do not need it for running BAMBOO.

In this tutorial I also describe how to use the SEDA event model which is implemented in BAMBOO. A more *modern* way to do that would be to use the Async implementation (see “Programmers Guide” on [?]).

2 Structure of a stage

This section will give you a rough overview of the elements of a stage and the usage of the config file. You will find more detailed explanation in section ?? where we will walk through the code of the example appended and explain the how and why more deeply, through that example.

2.1 First overview

A new stage must implement the `seda.sandStorm.api.EventHandlerIF` interface in order to act as a BAMBOO stage and be part of its event handling. There is also a class called `bamboo.util.StandardStage` you can inherit from. We will do this for our new stage, called *SimpleStage* (see Appendix ?? for the source code).

The stage can be seen as having three parts; each is referred to a different point in time, during the lifetime of a BAMBOO stage.

- `SimpleStage()` (the constructor)
- `init(ConfigDataIF config)`
- `handleEvent(QueueElementIF elem)`

The *Constructor* should be used to initialize the data of the stage (as you would do it in any Java programme). In BAMBOO you need to subscribe to events you want to hear from, so this is the place to fill the array structures provided by *StandardStage* with the events you need and want.

The registering of the events is done in the `init()` function. This function is called during startup, passing the configuration parameters to us. The registration is be done by *StandardStage*, so it is a good idea to call `super.init()` here. We also want to fetch all configuration parameters here, and it is a good place to register payloads for our outgoing messages.

`handleEvent()` is called from BAMBOO every time a event we registered to arrives and needs to be handled. This is the core of our stage.

There is another function called `destroy()` in `seda.sandStorm.api.EventHandlerIF`. It was meant to be a destructor but actually it is deprecated in BAMBOO and is never called.

2.2 The configuration file

One thing that BAMBOO inherits from Oceanstore is the SandStorm tool to parse the configuration file. The config file is XML-like and very easy to understand as you will see.

Listing 1: Our BAMBOO config file: simple.cfg

```

1<sandstorm>
  <global>
3  <initargs>
    node_id localhost:3200
5  </initargs>
  </global>
7 <stages>
  <Network>
9   class bamboo.network.Network
    <initargs>
11  </initargs>
  </Network>
13
  <Router>
```

```

15   class bamboo.router.Router
    <initargs>
17     gateway_count 1
      gateway_0 localhost:3200
19   </initargs>
  </Router>
21
  <SimpleStage>
23   class bamboo.SimpleStage
    <initargs>
25     debug_level 1
      mode sender
27   </initargs>
  </SimpleStage>
29 </stages>
</sandstorm>
```

As the format of the config file is easy to understand I will just explain the components of it and their meanings.

In line 2-6 you see the definition of global arguments. Every stage can access these arguments. You see here how to define the `node_id` for this stage, which consist of `hostname:portnumber`.

At Line 7 the stage configuration starts; we need the Network and the Router stages to send messages over the BAMBOO P2P system. You have to define at least one gateway to join the P2P network. In this example we join ourself, assuming that we are the first node in the network. If you are running more than one node you should add the node of the network you want to join, here. Take a look at the `init()` function of `bamboo.router.Router` in order to get an overview of the available Router options and how to use them.

As you can easily see, the definition of a new stage starts with a Tag containing the name of the stage (the classname) followed by the option class where the stage loader is told which class to load. For our stage *SimpleStage* we have two arguments called `mode`, which will later switch between sending messages or not, and `debug_level` which is used by *StandardStage* to switch on debugging messages.

3 Walk-through

In this section we will walk through the code of our *SimpleStage* (see Appendix ??) and explain the

important things in the example. This very simple stage will send (text-) messages to the node with the closest node-ID to zero. So fasten your seat-belt and let's go.

3.1 Class declaration and variables

Let's skip the first 16 lines, you should be familiar with that. As I mentioned before we extend from `bamboo.util.StandardStage` (it might be a good idea to risk a glimpse into its source code sometime later). `StandardStage` implements `seda.sandStorm.api.EventHandlerIF` and provides us a logging interface and some functions you maybe find useful when writing your own stage. When we use such a function from `StandardStage` I will point it out to you.

In lines 20 to 21 we help ourself to an application ID. It is used like a TCP/UDP port number in BAMBOO. When using the BAMBOO routing messages you have to specify the application ID you want to send to (will almost every time be your own ID). To get a unique number for our application (that is what a stage is, a BAMBOO application), we use a hash of our class name.

3.2 Our own Payload

We are now talking about lines 28 to 46. As you can see, we define a class named `Payload` that implements `ostore.util.QuickSerializable`. This is how to create a payload for your messages you want to send. The payload has to be serialized before sending and deserialization afterwards.

In line 39,

```
public void serialize(OutputBuffer b)
```

we have to add each variable we want to be transmitted to the output buffer. The counterpart is the constructor,

```
public Payload(InputBuffer buffer)
```

which deserializes the buffer back to the original variables. In our example there is just the message itself (a string) that we need to add to the buffer and save back from it.

Take a look into `ostore.util.InputBuffer` and `ostore.util.OutputBuffer` to see which types and objects are supported.

3.3 A new Event type

In line 49, we create a new message or event type (I think both notions are OK). To create one, we need to implement the `seda.sandStorm.api.QueueElementIF` interface which is in fact empty, so we do not need to implement any functions. Normally you want to add some content to the event like a payload (see section ??) and some other variables. In this case we use this event to alarm us after a certain time; and do not need any further functionality. There are plenty of examples in the BAMBOO source tree showing how to create your own event type.

3.4 The constructor

The important thing in our constructor is the setting up of the `event_types[]` array. In this array we save the classes of the events we want to listen to (these events will be forwarded to our `handleEvent()` function). `StandardStage` defines two more arrays for this purpose called `inb_msg_types[]` and `outb_msg_types[]`. As we do not need to use them at this point we will just cold-shoulder them for the moment. But I will talk about them in Appendix ??.

So you can see the three events we registered to here. The last one we already know: It is our own event `Alarm` we create a few lines above. The other two events are provided by BAMBOO itself, `seda.sandStorm.api.StagesInitializedSignal` is passed to us when all the stages of our node finished their `init()`, so it is a signal that we now can use the BAMBOO system.

`bamboo.api.BambooRouteDeliver` is the event we get when someone routed a message to our node through BAMBOO.

So remember: You need to add your events you want to receive into these arrays. We will see in section ?? what to do with these events and how the whole thing works.

Another important thing is shown in line 61. In order to use them we need to register our payloads to BAMBOO by passing the class as an argument:

```
register_type(Payload.class);
```

3.5 Initialization

When sandStorm starts all the stages it calls the `init()` function of each stage and passes the data from the configuration file to it.

The first thing we are doing is calling

```
super.init(config);
```

which passes the configuration data to the `init()` function of *StandardStage*. If you take a glimpse into *StandardStage* you will see that it does two things:

- Setting the debug-level for the logger
- Subscribing the events

The latter one is especially important for us – *StandardStage* subscribes our events to BAMBOO. I do not want to go more into detail: If you want or need to know more about that refer to the *StandardStage* code and Appendix ??.

Now it is time to get the config options we added to the config file in section ???. There are several functions to get the config data, in line 79 we use

```
config.get_string(config, "mode")
```

to retrieve a string from the option *mode*. There are more functions like that, e.g.

```
config.get_int(config, "<name>") or
```

```
config.get_boolean(config, "<name>"),
```

StandardStage will tell you more about these functions.

In the following lines we set the boolean `sender` to *true* if the config makes us to a sender, or *false* otherwise.

3.6 Event handling

`handleEvent(QueueElementIF elem)` is called every time an event shows up at our node. You will only get events you have subscribed to (see section ??) and – if it is a BAMBOO message like `bamboo.api.BambooRouteDeliver` – those that match your application ID, so the purpose of this function is to handle the event depending on which eventtype we received.

At this point a very important note: BAMBOO is a single-threaded event-driven application (see the “Programmers guide” on [?]). Because of this, you have to avoid blocking code in your event handler – even more, you should hurry up in your event handler because the whole (not only your stage, the

whole node) will be blocked until you finished your handling.

So let’s go through this step by step:

3.6.1 While not initialized

So let’s look at lines 90 to 111. As you will guess from the `if(initialized)` in line 90 the here occurring events will only appear during initialization.

One moment, we have done all initialization in the `init()` function already. Well, that is right, but there are still some things to do before our stage can to its regular work.

SandStorm will pass the *StagesInitializedSignal* to us as soon as every stage has finished its `init()` function. So at this point we can really use BAMBOO for sending messages. And we have to, in order to register our application ID (our “port”) to BAMBOO telling it that we want to receive events for that specific ID. This is been done by sending a `bamboo.api.BambooRouterAppReqReq`, passing as the first argument our application ID, and as the last one, a reference to our sink (that is our event queue).

By calling `dispatch(message)` the message is passed to the so-called *classifier*, which decides which stage should get this message (remember that the *StandardStage* subscribed to our events during `init()` from the classifier). We will use this function every time we want to send (dispatch) a message.

If the application registration succeeds (which it should) a `bamboo.api.BambooRouterAppReqResp` message will be passed to our sink. Now we are fully registered to the system, and set `initialized` to *true*. At line 109 you will see, that we put all occurring events during initialization into a `LinkedList`. Now after we are really up, we process all the events from this list.

In line 106 you will notice a new command:

```
dispatch_later(message, time);
```

It is almost the same `dispatch()` command as before but you can now specify a duration in milliseconds which tells the classifier how long to wait till delivering the message. We use this command to set a alarm timer to awake us after 10 seconds.

3.6.2 BambooRouteDeliver

When a message was sent to us routed by BAMBOO (means using the `bamboo.api.BambooRouteInit` message which we will get to know in section ??), we will receive a `BambooRouteDeliver` message.

In line 116 we cast the event to get access to its data. As you see we assume that there is our *Payload* attached as a payload and print out the string saved in it. There are some more fields in this message: Please take a look into the source for more information.

3.6.3 Alarm

When we are woken by our own *Alarm* message we send a new text message through BAMBOO to the node with the lowest available node ID, and start another alarm timer. So the lines 132 and 135 should be familiar, but not line 126 and the following.

In these line you see how to let BAMBOO route a message for you. To tell BAMBOO that, you have to use a `bamboo.api.BambooRouteInit` message. The arguments are:

- destination node ID
- application ID
- use upcalls?
- use iterative routing (or recursive)?
- the payload to be transmitted

As you should already know BAMBOO uses 160 bit node IDs as addresses (like Pastry). To save such a long number in Java we have to use a `BigInteger` object. In this example we route the message to the nodeID 0 (zero).As you should know, when this node does not exists BAMBOO routes the message to the node with the closest node ID.

The thing with the application ID is known since section ??, so no more words about that.

If you pass *true* as the next argument, BAMBOO will make upcalls to every interim node on the way of the message to its destination. These nodes will receive a `bamboo.api.BambooRouteUpcall` event (if they subscribed to this event and if they have the same application ID of course) and can

change the message or react on it. And they need to continue the routing by dispatching a `bamboo.api.BambooRouteContinue`. I will not describe more of this mechanism: It is easy to understand and familiar in every P2P system I know. Take a glimpse into the two mentioned classes for more information.

The switch for choosing between iterative and recursive routing is deprecated, so just say *false* here.

Last but not least, we add the payload we want to transmit. That's it, so easy to send a message over BAMBOO.

4 Finally

Finally you want to run your stage. So compile *SimpleStage* and save the `simple.cfg` file to the appropriate place. Change (if not already done) to the BAMBOO directory and type:

```
./bin/run-java bamboo.lss .DustDevil\  
simple.cfg
```

This will start *DustDevil* which uses *SandStorm* to parse the configuration file “`simple.cfg`” and starting the stages included in the it. You should see the in the output how the messages are sent and received.

You may start more nodes by adapting “`simple.cfg`”.

So that's it: If you understand the things we used in this tutorial you are ready to write your own one or start reading through the code of the other stages shipped with BAMBOO. You will find these structure in almost every stage (beside the stages that uses the ASync event model).

In Appendix ?? you will find instructions how to send messages direct to a node without routing them through BAMBOO. Appendix ?? contains the full source code for the *SimpleStage*.

References

- [1] <http://www.bamboo-dht.org>
- [2] <http://research.microsoft.com/~antr/Pastry/>

[3] <http://www.cs.berkeley.edu/~ravenben/tapestry/>

[4] <http://oceanstore.org/publications/papers/abstracts/iptps03-api.html>

Appendix

A Sending messages direct to a node

Sometimes you might not want to use BAMBOO to route a message to a node, but instead send it “direct” to a specified address (e.g. an IP address), and not to its closest node (I set the word direct into quotation marks because the message will still be routed if the destination is not physical connected to our node). For this reason you want to use `ostore.network.NetworkMessage`.

Let’s have a quick look at `ostore.util.NodeId`. `NetworkMessage` uses this class for addresses. `StandardStage` creates one for your node, the variable name is `my_node_id`. Take a look into `ostore.util.NodeId`: To create a new `NodeId` object you need just the IP address and the port number. You will need to have such an object for the destination node you want to send your message to.

A.1 The Network stage

So let’s go through it step by step:

First, create a new `NetworkMessage` object, e.g.:

```
NodeId dest =
  new NodeId(3200, InetAddress.
    getByName("localhost"));
NetworkMessage nm =
  new NetworkMessage(dest, false);
```

If you add your sink (`StandardStage` saves it to `my_sink`) to `comp-q`, the Network stage will send you a `ostore.network.NetworkMessageResult` which will tell you if transmission was successful or not. If you put something in `user_data` this will also be passed to you together with the `NetworkMessageResult` event.

Dispatch the event, and that’s it – you’ve sent the message directly. If you want to send any payload you will have to create a

new event type extending from `NetworkMessage` and adding a variable for the payload (of type `ostore.util.QuickSerializable`).

A.2 Again: Event types

So how does this work? For explanation we have to step back to section ?? just for a moment where I mentioned that there are two more arrays for events called `outb_event_types[]` and `inb_event_types[]`.

When `StandardStage` subscribes to these messages it also adds a filter to each event listed in these arrays. It checks that there is a boolean called `inbound` in each event and it tells the classifier that this stage only wants to receive these events if `inbound` is `true` (for `inb_event_types[]`) or `false` in the other case (for `outb_event_types[]`). This is one possible filter (like the one you have with BAMBOO messages that are filtered by your application ID), take a look into `bamboo.util.StandardStage` for an example how to use filters.

The thing with the Network stage is that it listens to `NetworkMessages` with `inbound` set to `false` and sends them to the destination represented by the `NodeId` object.

A.3 Receiving a NetworkMessage

So we sent a `NetworkMessage` but we have not done anything to receive them (remember we have to subscribe to each message we want to listen to).

The Network stage sends the message to the Network stage of the destination node. This stage then sets `inbound` to `true` and sets the new destination to its own node ID and dispatches it. Local stages that subscribe to this type of message will now receive it:

```
inb_event_types [] =
  new Class [] { NetworkMessage.class }
```

The original sender can be found in the variable named `sender` (not in `source!`).

A.4 NetworkMessageResult

I mentioned this before: If you provide your sink to the `NetworkMessage` it will send you

back an `ostore.network.NetworkMessageResult` event which will inform you about successful or unsuccessful transmission. Therefore check the variable `success`, the `user_data` is also passed back to you in this message so that you can identify the message that failed. Of course you need to subscribe to this event type, so add it to `event_types[]`.

B The sourcecode of SimpleStage

```
1 package bamboo;

3 import java.math.BigInteger;
  import java.util.LinkedList;

  import ostore.util.InputBuffer;
  import ostore.util.OutputBuffer;
8 import ostore.util.QuickSerializable;

  import bamboo.api.*;
  import bamboo.util.StandardStage;

13 import seda.sandStorm.api.ConfigDataIF;
  import seda.sandStorm.api.EventHandlerException;
  import seda.sandStorm.api.QueueElementIF;
  import seda.sandStorm.api.StagesInitializedSignal;

18 public class SimpleStage extends StandardStage{

    protected static final long app_id =
        bamboo.router.Router.app_id(SimpleStage.class);

23 private boolean sender = false;

    protected boolean initialized = false;
    protected LinkedList wait_q = new LinkedList();

28 protected static class Payload implements QuickSerializable{
    public String message;

    public Payload(String m){
        message = m;
33     }

    public Payload(InputBuffer b){
        message = b.nextString();
    }

38     public void serialize(OutputBuffer b){
        b.add(message);
    }

43     public String toString(){
        return "Payload with message: " + message;
    }
}

48 /** Just for alarming with no further functionality */
  protected static class Alarm implements QueueElementIF{

    /**
```

```

53  * Constructor <br>
    * Calling the constructor of our super class, register
    * all events we want to listen to.<br>
    * You should also register your own payloads here.
    */
public SimpleStage() throws Exception {
58     super(); // Sets up the logger

    // Register payloads
    ostore.util.TypeTable.register_type(Payload.class);

63     // Bamboo events we wanna listen to
    event_types = new Class[] {
        StagesInitializedSignal.class,
        BambooRouteDeliver.class,
        Alarm.class
68     };
}

/**
73  * Initialize our stage. We get the parsed Config data
    * from sandstorm and may extract our own config options
    * beside the global ones. <br>
    */
public void init(ConfigDataIF config) throws Exception{
78     super.init(config);

    String mode = config_get_string(config, "mode");
    if(mode != null && mode.equals("sender"))
        sender = true; // sender mode
    else
83     sender = false; // default
}

public void handleEvent(QueueElementIF elem) {
88     logger.debug("Got event " + elem);

    // If startup
    if(!initialized){
        // This one is sent when setting up
        if(elem instanceof StagesInitializedSignal){
93             // Request registration for this app
            dispatch(new BambooRouterAppRegReq(
                app_id, false, false, false, my_sink));
        }
        // OK, we are now registered to Bamboo
98     else if(elem instanceof BambooRouterAppRegResp){
        // handle pending events
        initialized = true;
        while(!wait_q.isEmpty())
            handleEvent((QueueElementIF) wait_q.removeFirst());
103     // Dispatch in 10s a Alarm msg to stages on this node
    if(sender)

```

```

classifier.dispatch_later(new Alarm(), 10000);
    }
108     // For pending events before we are registered
    else
        wait_q.addLast(elem);
    }
    // Normal operational mode
113 else{
    // Event that we got a message delivered
    if(elem instanceof BambooRouteDeliver){
        BambooRouteDeliver deliver = (BambooRouteDeliver) elem;
        Payload pay = (Payload) deliver.payload;
118         logger.info("Message is: " + pay.toString());
    }
    else if(elem instanceof Alarm){
        // Create a new message to be send over bamboo, only
        // stages with the same app_id will get this message!
123         String msg =
            "This message should be send over" +
            "Bamboo to the node with the smallest node ID.";
        BambooRouteInit init = new BambooRouteInit(
            BigInteger.ZERO,
128             app_id,
            false, false,
            new Payload(msg));
        // Send the message to the sink.
        dispatch(init);
133
        // Dispatch in 10s a new Alarm msg to the stages on this node
        classifier.dispatch_later(new Alarm(), 10000);
    }
    else{
138         BUG("Event " + elem + " unknown.");
    }
    }
}
143 }

```