TBN Tutorial

Cosmin Arad
cosmin@sics.se

January 29, 2008

1 Introduction

TBN is a software architecture for building component-oriented, event-based, concurrent systems that are dynamically reconfigurable. This tutorial describes the TBN architecture and how to use the Java implementation of it, together with some built-in components (e.g. timers maintenance, network communication, failure detection, etc.).

Systems built using the TBN architecture are structured as reactive *components* that interact with each other by exchanging *events*. Components are connected to each other by *channels*. Component interactions follow a publish-subscribe mechanism, i.e. components publish events into channels and subscribe to events published into channels.

Components are first-class objects that can be dynamically created or destroyed by *factories*. Components interactions can also be altered while the system is running. Thus, the system can be updated without being restarted.

Let us now take a closer look at the system architecture.

2 System Architecture

There are four entities in the TBN architecture: components, events, channels, and factories. We describe each one in detail in the following sections, by referring to the Java implementation of TBN.

2.1 Components

A component is an object. It contains some internal[1] variables that capture its state. Typically, components are passive objects[2], i.e. they do not have threads of control of their own. Components execute as a result of receiving an event. Thus, we say that components are reactive. When a component receives an event it handles it by executing an *event handler*. An event handler is a method that takes the received event as an argument. The internal variables of a component can only be accessed or changed by event handlers. A component can handle any type of event for which it has a declared event handler.

A pool of worker threads is shared by all components in the system. When a component needs to execute an event handler in response to receiving an event, it will use a worker thread from the pool to execute the event handler on its behalf. Multiple components can execute concurrently in the system. However, one component will only execute one event handler at a time. We say that event handlers are atomic. Thus, the internal variables of the component are protected from concurrent accesses.

While executing an event handler, a component can trigger new events for other components. These events are published into channels and are received by all components that are subscribed to the respective channels.

2.2 Events

An event is an object with a set of attributes. There can by many types of events in the system, each event type being a Java type (class) that implements the interface `tbn.api.Event`. When a component triggers an event, an event instance of the respective event type is created.

Upon triggering an event, a component can specify one of three priority levels for the event: low, normal, or high. In general, high priority events are executed before normal priority events, and normal priority events are executed before low priority ones. Under heavy load, starvation of low priority events is prevented by a fairness rule: no more than f high priority events are executed once a normal priority event has been triggered, and no more than f normal priority events are executed once a low priority event has been triggered. We call f the fairness factor.

As an example, timers related events are given high priority.

[1] encapsulated, not visible or accessible by other components.

[2] we discuss exceptions to this in Sections 4.1, 4.2, and 4.4.

2.3 Channels

Channels are a virtual communication medium between components. Components can publish events into channels and can subscribe their event handlers to channels. An event that is published into a channel, is delivered to all components that have subscribed to that channel. Thus, component interaction occurs by passing events through channels.

We can say that two interacting components are linked together by a channel. If component A is to pass an event to component B, A has to *publish* the event into a channel to which B has previously *subscribed*. Some components should be reusable, independent of other components, therefore they should not have any hardwired links to other components. Thus, a component's functionality and its connections are orthogonal properties. We view these two as being provided by two different roles: a *developer* who writes the component functionality, and a system *configurator* who puts together and links multiple components. Given this, a component's implementation is unaware[3] of the channels though which it receives events and into which it publishes events.

It is the configurator who subscribes a component to a channel and who binds event types (triggered by the component) to a channel. Thus a component's event handler can just trigger an event without knowing the channel into which the event is being published.

A channel has an associated set of event types that can be published into it. Type checks are made when a component subscribes an event handler to a channel to verify that the handler can accept any of the event types associated to the channel. Also, when an event type is being bound for a component to a channel, it is checked whether the channel supports the respective event type.

Channels preserve the order of events that are published into them. This means that each subscriber to a channel receives the events that are published into that channel in the same order in which they were published. We say that the channels are FIFO.

2.4 Factories

Factories are managers of components of a certain type. There is a component factory for each component type, which can create, destroy, and find by name components of that respective type. Thus, a factory is also a registry of component instances.

[3] although it can become aware.

Because dynamic reconfiguration of the system should be possible from outside the address space of the system, the components and channels in the system should be addressable by unique names. Each component instance has a unique name within its factory and each factory has a globally unique name. Channels have globally unique names.

2.5 Event handlers

An event handler is a component method that takes one argument, the event just received by the component. Within one component, event handlers execute atomically but event handlers of different components can execute concurrently. Event handlers are executed by workers from a system thread pool.

Event handlers may have associated guards. A guard is a boolean condition on the component's internal variables. It is implemented as a Java method that takes no arguments and returns a boolean value. If an event handler has an associated guard, the condition is evaluated before executing the handler. If the condition evaluates to `true` the handler is executed. Otherwise, the event is queued into a local handler queue. In other words, guarded event handlers are only executed when their guard allows it.

After an event handler finishes executing, it has potentially changed the internal variables of the component, thus for all guarded event handlers that have non-empty queues, the conditions are reevaluated. If a condition evaluates to true, the first event queued in that handler is executed.

2.6 Event scheduling

Events that are triggered by the execution of an event handler are not made visible to the system before the triggering event handler finishes executing. When the triggering event handler terminates, for all events that it triggered, the system determines all the subscriber components and creates work pieces for the worker thread. A work piece is a <component, event handler, event> triple. The work pieces are tagged with the event's priority and are added to a fair[4] priority queue where from the work pieces are taken by the worker threads and executed.

[4] see Section 2.2

2.7 Component and channel life-cycle

We mentioned that the system is dynamically reconfigurable, i.e. components can be added and removed at runtime, channels can be created and destroyed, and components' subscriptions and bindings can be changed while the system is running.

The system allows for replacing an old component with a new one, without losing any events that were meant to be received by the old component. This is achieved by introducing the notion of life-cycle for components and channels. Components and channels can be either *active* or *passive*. Initially, components and channels are passive. They can be made active by means of a *start* operation, and they can be made passive again by means of a *stop* operation.

A component that is active can execute events, but its bindings and subscriptions cannot be changed. Vice-versa, if a component is passive, its wirings can be changed but it does not execute events.

Events published into an active channel are immediately delivered to the subscriber components. Events published into a passive channel are queued in a channel event queue. To avoid the loss of events (by delivering events to passive components) the system allows stopping a component only when all the channels to which the component has subscribed event handlers are passive.

3 Hello World! Example

Let us now look at a TBN HelloWorld application. We have two components: a `WorldComponent` named 'WorldFactory:World' and a `MyComponent` named 'MyFactory:Me'. We have two channels: a 'HelloChannel', carrying events of type `HelloEvent` and a 'ResponseChannel' carrying events of type `ResponseEvent`. The 'World' component is subscribed to the 'HelloChannel' and the 'Me' component is subscribed to the 'ResponseChannel'. The 'World' component is bound to publish triggered events of type `ResponseEvent` into the 'ResponseChannel' and the 'Me' component is bound to publish triggered events of type `HelloEvent` into the 'HelloChannel'. When started, component 'Me' triggers a `HelloEvent`. This event is handled by the 'World' component, using its `handleHelloEvent` handler, which triggers a `ResponseEvent`, received by the 'Me' component and handled by its `handleResponseEvent` handler. The Java code for the `WorldComponent` is shown in Figure 1.

The `WorldComponent` contains no internal variables and an event handler

```
public class WorldComponent {
   private final Component component;
      public WorldComponent(Component component) {
      this.component = component;
   }
   public void handleHelloEvent(HelloEvent event) {
      String message = event.getMessage();
      System.out.println("World: I got message: \"" + message + "\"");
      component.raiseEvent(new ResponseEvent("Hi there!"));
      System.out.println("World: I replied: Hi there!");
   }
}
```

Figure 1: The Java code for `WorldComponent`.

that handles events of type `HelloEvent`. Every component has to have a reference to a component handler that it can use to trigger events. This component handler is referenced through the interface `tbn.api.Component`. It is mandatory for a component to have a public constructor that take a `tbn.api.Component` as an argument. The Java code for the `MyComponent` is shown in Figure 2.

```
public class MyComponent {
private final Component component;
   public MyComponent(Component component) {
      this.component = component;
   }
   public void start() {
      component.raiseEvent(new HelloEvent("Hello"));
      System.out.println("Me: I said Hello to the World");
   }
   public void handleResponseEvent(ResponseEvent event) {
      String message = event.getMessage();
      System.out.println("Me: I got message: \"" + message + "\"");
   }
}
```

Figure 2: The Java code for `MyComponent`.

The main difference from the `WorldComponent` is that `MyComponent` declares a start method that takes no arguments. This method is called when the component is started.

The Java code for `HelloEvent` is given in Figure 3. The code for `ResponseEvent` is very similar.

The Java code for the main program (which takes the configurator role in this case) is given in Figure 4.

```
public class HelloEvent implements Event {
private final String message;
   public HelloEvent(String message) {
      this.message = message;
   }
   public String getMessage() {
      return message;
   }
}
```

Figure 3: The Java code for `HelloEvent`.

First, a reference to the TBN system is created, by calling `TBN.getSystem()`. This is then used to create factories for `WorldComponent` and for `MyComponent`, and to create the channels with names 'HelloChannel' and 'ResponseChannel'. The two factories are used to create components 'WorldFactory:World' and 'MyFactory:Me', respectively.

After the components and the channels have been created, the components are subscribed and bound to the channels, as described earlier. Once the system configuration is done, the components and the channels are started. Finally, a reconfiguration server is started. The reconfiguration server exposes a configuration interface to the system to an external configurator that runs a TBN configuration client (see Section 4.5).

At this point, the main program thread exits but the whole system is executed onward by the worker thread in the thread pool, that was internally created when the system was created (by the first call to `TBN.getSystem()`). An alternative main program, equivalent to the one in Figure 4, is given in Figure 5.

This program builds the exact same system, by reading the system's configuration from an architecture description file (`hello.xml`). The architecture description file is an XML file declaring the factories, components, and channels of a system, the bindings and subscriptions of components, and the startup sequence of components and channels. The contents of the `hello.xml` configuration file are given in Figure 6.

4 TBN Library Components

Some useful library components and tools are provided for building distributed systems. These include: a timer component that provides an alarm service for other components, a communication component that provides network communication between processes of a distributed system, and a

```
public class HelloMain {
   public static void main(String[] args) throws Exception {
      TBNSystem sys = TBN.getSystem();
      Factory myFactory = sys.createComponentFactory("MyFactory", "MyComponent");
      Factory worldFactory = sys.createComponentFactory("WorldFactory", "WorldComponent");
      Component myComponent = myFactory.create("Me");
      Component worldComponent = worldFactory.create("World");

      Channel helloChannel = sys.createChannel("HelloChannel");
      helloChannel.addEventType(HelloEvent.class);
      Channel responseChannel = sys.createChannel("ResponseChannel");
      responseChannel.addEventType(ResponseEvent.class);

      myComponent.subscribe("ResponseChannel", "handleResponseEvent");
      worldComponent.subscribe("HelloChannel", "handleHelloEvent");
      myComponent.bindOutputEvent(HelloEvent.class, "HelloChannel");
      worldComponent.bindOutputEvent(ResponseEvent.class, "ResponseChannel");

      myComponent.start();
      worldComponent.start();
      helloChannel.start();
      responseChannel.start();
      sys.startReconfigurationServer();
   }
}
```

Figure 4: The Java code for `HelloMain`.

failure detector component that monitors the liveness of other processes of a distributed system.

4.1 Timer Component

The timer component allows other components to set one-time timers. It interacts with its client components through events. It accepts a `TimerEvent` triggered by client components to either set a timer or cancel a set timer. In the `TimerEvent`, the client component specifies a notification event that

```
public class HelloMainADL {
   public static void main(String[] args)) throws Exception {
      TBNSystem sys = TBN.getSystem();
      sys.buildSystem(HelloMainADL.class.getResource("hello.xml").getPath());
      sys.startReconfigurationServer();
   }
}
```

Figure 5: The Java code for `HelloMainXML`.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:configuration xmlns:tns="http://www.sics.se/adl">
   <tns:factories>
      <tns:factory>
         <tns:factoryName>WorldFactory</tns:factoryName>
         <tns:componentClass>tbn.examples.hello.WorldComponent</tns:componentClass>
      </tns:factory>
      <tns:factory>
         <tns:factoryName>MyFactory</tns:factoryName>
         <tns:componentClass>tbn.examples.hello.MyComponent</tns:componentClass>
      </tns:factory>
   </tns:factories>
   <tns:components>
      <tns:component>
         <tns:factoryName>WorldFactory</tns:factoryName>
         <tns:componentName>World</tns:componentName>
      </tns:component>
      <tns:component>
         <tns:factoryName>MyFactory</tns:factoryName>
         <tns:componentName>Me</tns:componentName>
      </tns:component>
   </tns:components>
   <tns:channels>
      <tns:channel>
         <tns:channelName>ResponseChannel</tns:channelName>
         <tns:eventClass>tbn.examples.hello.ResponseEvent</tns:eventClass>
      </tns:channel>
      <tns:channel>
         <tns:channelName>HelloChannel</tns:channelName>
         <tns:eventClass>tbn.examples.hello.HelloEvent</tns:eventClass>
      </tns:channel>
   </tns:channels>
   <tns:bindings>
      <tns:binding>
         <tns:componentName>WorldFactory:World</tns:componentName>
         <tns:eventClass>tbn.examples.hello.ResponseEvent</tns:eventClass>
         <tns:channelName>ResponseChannel</tns:channelName>
      </tns:binding>
      <tns:binding>
         <tns:componentName>MyFactory:Me</tns:componentName>
         <tns:eventClass>tbn.examples.hello.HelloEvent</tns:eventClass>
         <tns:channelName>HelloChannel</tns:channelName>
      </tns:binding>
   </tns:bindings>
   <tns:subscriptions>
      <tns:subscription>
         <tns:componentName>WorldFactory:World</tns:componentName>
         <tns:channelName>HelloChannel</tns:channelName>
         <tns:handlerName>handleHelloEvent</tns:handlerName>
      </tns:subscription>
      <tns:subscription>
         <tns:componentName>MyFactory:Me</tns:componentName>
         <tns:channelName>ResponseChannel</tns:channelName>
         <tns:handlerName>handleResponseEvent</tns:handlerName>
      </tns:subscription>
   </tns:subscriptions>
   <tns:startSequence>
      <tns:startComponents>
         <tns:componentName>WorldFactory:World</tns:componentName>
         <tns:componentName>MyFactory:Me</tns:componentName>
      </tns:startComponents>
      <tns:startChannels>
         <tns:channelName>ResponseChannel</tns:channelName>
         <tns:channelName>HelloChannel</tns:channelName>
      </tns:startChannels>
   </tns:startSequence>
</tns:configuration>
```

Figure 6: The architecture description for the HelloWorld! system.

the timer component should trigger for the client component, and the time duration after which this notification event should be triggered. Notification events specified by client components must extend the `TimerExpiredEvent` class.

Components that use the services of the timer component should instantiate and use a `TimerHandler` object. The `TimerHandler` provides some useful wrappers for interacting with the timer component. To set a timer, a component calls the `startTimer` method of the timer handler, providing the notification event that it wants to receive on timeout, the time duration before the timeout, and the name of the handler that will handle the timeout notification event. The `startTimer` method properly creates and triggers (with high priority) a `TimerEvent` for the timer component. The timer event is given a unique identifier. This unique identifier is returned by the `startTimer` method and the component can use it to cancel the timer by passing it as an argument to the `stopTimer` method of `TimerHandler`.

The timer handler keeps track of all timers that have been set by the component and that have not yet been canceled or expired. These are called outstanding timers. Because the client component and the timer component execute concurrently it is possible that the client component cancels a timer after the timer has expired and the client component receives the timeout notification event, even though it has canceled the timer. Therefore, whenever a component handles a timeout notification event, it should check with the timer handler whether the timer is still outstanding or not. To prevent any inconsistent behavior, a component should only handle timeout notification events corresponding to outstanding timers.

For example, consider a component that sends a message to another process and waits for a reply message. If the reply is received within a specified timeout, the component executes action A. If the reply is not received within the timeout the component executes action B, where actions A and B are mutually exclusive. So, the component sets a timer to expire after the specified timeout, and deliver a notification event to the component on expiration. If the component receives the reply message it will cancel the timer and execute action A. It is possible that the timer component has triggered the expiration notification to the client component just before the client component canceled the timer. In this case, the client component will receive a timer expired notification even though it canceled the timer, so when it receives it, it has to check with the timer handler if the corresponding timer is still outstanding (by calling the method `isOutstanding()`). If it does not check this, in our example, the client component executes both actions A and B, which was not intended.

In summary, a component should use a `TimerHandler` object to start and cancel timers, and the `TimerExpiredEvent` as a timeout notification event. It should extend this if it needs to associate more state with the notification.

The timer component differs from a typical component, in the fact that it uses a separate thread for timers management.

4.2 Communication Component

The communication component enables other components in the system to send and receive messages to/from remote components in other processes. It allows communication over the TCP or the UDP network transport protocols.

A process is identified by a `NodeReference` which is a triple containing the IP address of the machine where the process is running, the TCP/UDP port on which the communication component listens for incoming connections from its counterparts in other processes, and a process identifier that is application specific (represented as a `BigInteger)`. When an application uses the communication component it must specify its desired node reference by calling the static method `setThisNodeReference` of the `NodeReference` class. The node reference specified in this way is used by the local communication component to listen for incoming connections from the communication components in other processes.

The communication component accepts events of type `MessageEvent`. Client components that need to send and receive messages should declare the messages as objects that extend the `MessageEvent` class, which contains node references of the source node and the destination node of the message. To send an event, a client component needs only to trigger it. Useful wrappers for sending messages are provided by a `MessageHandler` object. Thus, any component that needs to send messages over the network should instantiate a `MessageHandler` and call its send method passing the message and the destination node reference as arguments. The `MessageHandler` will set the source node reference as the local node's reference and will trigger the message event for the communication component. Optionally, the client component can pass an argument to the send method to specify the type of network transport protocol (TCP or UDP) that the communication component should use for sending the message. By default, the communication component uses UDP.

The communication component automatically manages network connections to other processes and automatically marshals messages before sending

them and unmarshals them after receiving them. When the communication component receives a message from the network it unmarshals it and trigger it as an event. Thus, some local component that has subscribed for it will handle it.

The communication component differs from a typical component, in the fact that it uses four separate threads for accepting and initiating UDP and TCP connections, respectively. This stems from using the MINA communication framework.

4.3 Failure-detector Component

The failure detector component implements an eventually perfect failure detector, and offers client components the service of monitoring the liveness of remote processes. A client component can ask the failure detector component to start or stop monitoring a certain process identified by its node reference, by means of triggering a `StartMonitoringNode` or a `StopMonitoringNode` event.

The failure detector component keeps track of all the monitoring requests it receives from client components and ignores requests to stop monitoring a certain process that come from a component that has not previously requested the monitoring of that respective process. Thus, many client components can safely use the services of the failure detector component without interfering with each other.

When the failure detector suspects that a remote process has crashed it triggers a `FailureEvent` with a SUSPECT failure type, to all components that have requested the monitoring of that respective process. When it detects that the process is alive again, it triggers a `FailureEvent` with a REVISE failure type, to all interested components.

A client component that needs to use failure detection services should instantiate a `FailureDetectorHandler` object that provides useful wrappers for triggering the `StartMonitoringNode` and `StopMonitoringNode` request events. The `FailureDetectorHandler` properly creates and triggers these events, such as to specify the requesting component and the channel into which failure notifications raised by the failure detector component should be published.

The failure detector component is a reactive component (it does not have any threads of its own) and it uses the communication component to send and receive ping and pong messages, as part of its implementation.

4.4 Application/user Component

We call an application component any component that uses a separate thread for interfacing with the user, and triggers events in the system as a response to commands given by the user.

4.5 TBN Inspector

The TBN inspector is a GUI application that connects to a remote TBN process that has started a reconfiguration server. The TBN inspector presents the user with a graphical view of the remote system's architecture (components, channels, subscriptions, bindings).