The MimiCry Framework

Marc-Christian Schulze

July 9, 2013

Contents

Chapter 1

Introduction

Mimicry is a non-intrusive network simulation framework for Java applications. Various other frameworks are currently available such as Tiny Sim, JNS, DSSim, Java Network Simulator, Peerfect and ns2. Typically these frameworks provide APIs which can be used to write prototype implementations of network protocols which then can be tested within a controlled environment. These kind of simulators typically provide a discrete behavior of the simulation. However, they are actually simulating prototypes.

The Mimicry framework does not require to compile the simulated applications to any part of the simulator's API. Instead it uses byte-code manipulation to load the application under test and intercepts all interactions with the JVM. This enables us to run the actual production ready code within the simulator.

1.1 Download and Compile Mimicry

Mimicry is currently only available via the Git repository so you need to download and compile the sources on your machine. In order to do so you first need to check that you've installed all prerequisites:

- JDK 7
- Maven
- Git

To check the latest source code out of git you need to run the following command in a shell:

```
git clone https://code.google.com/p/mimicry
```

Now you can compile the Mimicry sources using Maven:

```
cd mimicry/parent
mvn clean install -DskipTests
```

Note: You need to skip the unit tests since some of them are currently failing due to a bug. After the successful compilation a zip will be created in the target directory of the distribution project:

`mimicry/mimicry-distribution/target`

Extract this archive to any location of your hard drive and make the *mimicry.sh* shell script executable:

```
chmod +x mimicry.sh
```

1.2 Prepare an Application for Simulation

In order to run your application Mimicry needs some information about where your binaries are located, how the classpath has to look like, how your main class is named, etc. This information is internally managed in a so-called *ApplicationDescriptor*. Once you've setup such a descriptor Mimicry will be able to load and run your application. In addition to the applications you also to setup the network itself, e.g. create nodes, define event stacks, etc. This is done in a Groovy-Script that is used to bootstrap and control the simulation. A simple script for setting up a network with a single node and application could look like this:

```
// Initialize the network
NetworkConfiguration netCfg = [
  clockType: ClockType.REALTIME,
  initialTimeMillis: 0
]
network.init(netCfg)

// Define an EventStack
EventHandlerConfiguration[] eventStack =
[
  [
    className: "com.gc.mimicry.plugin.tcp.PortManager"
  ],
  [
    className: "com.gc.mimicry.plugin.tcp.SimpleTCPDataTransport"
  ],
  [
    className: "com.gc.mimicry.plugin.tcp.TCPConnectionManager"
  ]
]

// Create a custom application descriptor
builder = new ApplicationDescriptorBuilder("My-Application")
builder.with {
  withMainClass( "org.example.MainClass" )
  withCommandLine( "some parameters" )
  withClassPath( "my-jar.jar" )
  withClassPath( "some-dependency.jar" )
}
applicationDesc = builder.build()

// Define how the node should be named and which stack to use
nodeCfg = new NodeConfiguration("ServerNode")
nodeCfg.eventStack.addAll( eventStack )
```

```
35
36 // Create the actual node and application instances within the network
37 nodeRef = network.spawnNode(nodeCfg)
38 appRef = network.spawnApplication(nodeRef, applicationDesc)
39
40 // Start the main thread of the application
41 network.startApplication(appRef)
42
43 // Start the timeline of the simulation
44 // The multiplier of 1.0 indicates that the simulation is running
45 // as fast as the system time
46 network.getClock().start(1.0)
```

As you can see in the listing above the simulation setup consists of the following basic steps:

- Initialize the Network
- Define EventStack and ApplicationDescriptors
- Create Nodes and spawn Applications
- Start the Timeline

1.3 Run the first Simulation

Mimicry ships with some predefined applications and simulation scripts. You can download them from

`https://code.google.com/p/mimicry/downloads`

For illustration we'll use the PingPong-Example which runs two application instances sending each other messages using a TCP/IP connection. After you've downloaded the *example-PingPong.zip* you need to extract its content into the installation directory (where you did extract the compiled mimicry zip file). Open a shell in that directory and run the following command:

```
./mimicry.sh -mainScript pingpong.groovy
```

This should bring up two console windows where in the first the server and in the second the client is writing its stdout to.

Chapter 2

Framework Architecture

This chapter explains the architecture of the Mimicry framework showing how all the parts work together.

2.1 Class Loading and Byte-Code Manipulation

The core of the Mimicry framework is build by the internal used class loading mechanism in combination with byte-code manipulation at load-time. Using the custom class loading mechanism Mimicry isolates each simulated application from others and the actual framework. The byte-code manipulation is used to intercept all interactions of the simulated applications with the JVM. The application's byte-code is loaded in two phases:

1. Code Loading and Loop Interception

   The actual class files are read from the hard drive using the Soot framework which transforms the byte-code into an intermediate representation that can be analyzed and modified. Leveraging the capabilities of Soot, loops are detected within the byte-code and a static method invocation added which is used for the life-cycle management later on. The resulting intermediate model is then transformed back to Java byte-code which is finally passed to the second phase.

2. Java API Interception

   The second phase is realized using AspectJ to intercept the Java API. A specialized derivate of the *WeavingURLClassLoader* is used to pass the modified byte-code to AspectJ which applies all aspects of Mimicry to the application's code.

Both above-mentioned phases are implemented in a single class loader which is instantiated per simulated application instance. This isolates the instances from each other and allows to load classes multiple times (for each application) at the same time into the JVM. This approach is comparable to the one used in OSGi. The entire hierarchy of the class loaders used is depicted in Figure 2.1.

Each *WeavingClassLoader* is responsibly for loading all application code and weaving it using Mimicry's aspects. On the next higher level a child-first or parent-last class loader is placed which prevents the *WeavingClassLoader*

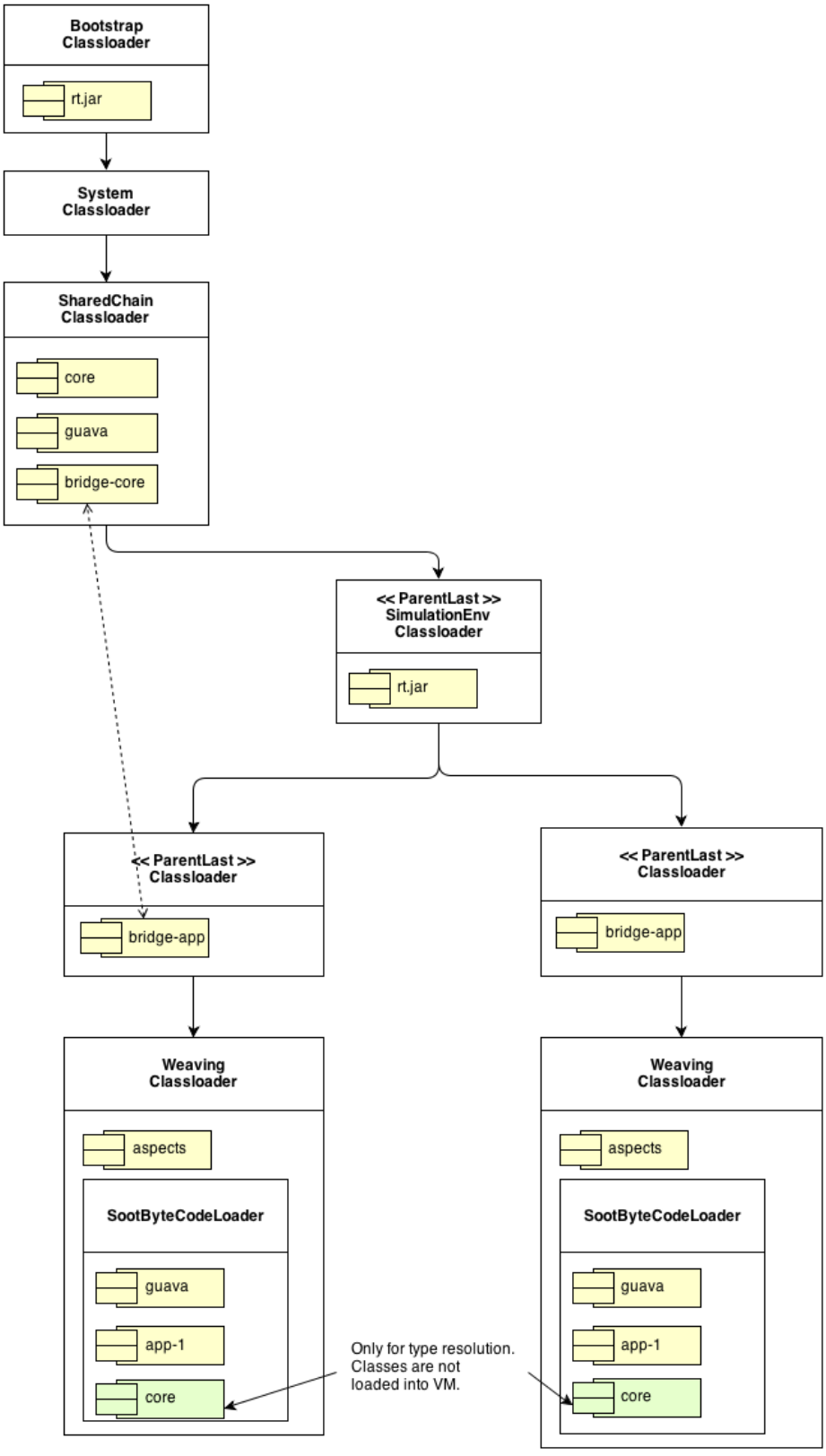

Figure 2.1: The hierarchy of the ClassLoaders

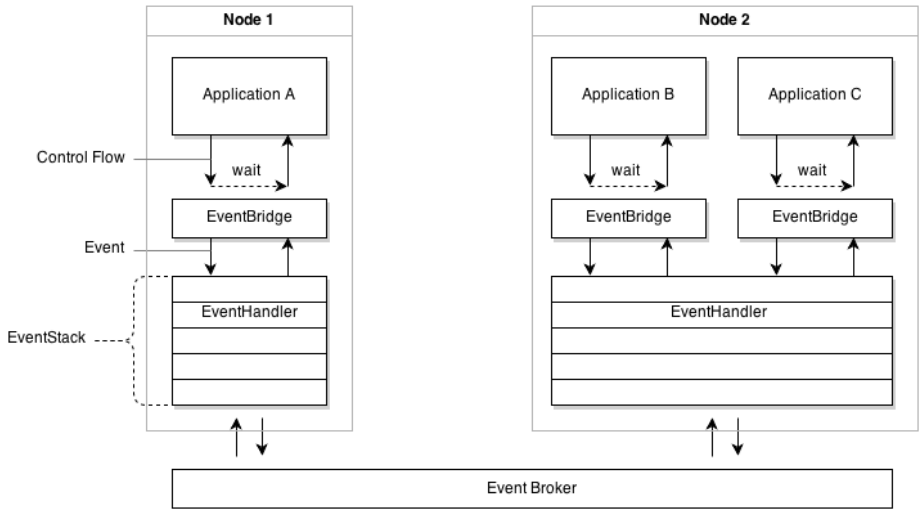

Figure 2.2: Architecture of the Event Engine

from requesting application classes from the parent, which might also be able to load for instance the Guava library (since it's used internally). Those class loader instances are the actual border among the applications and the framework. A special package, called the Simulator Bridge, is located. This bridge is used by Mimicry's aspects to communicate with the simulation engine placed in the *SharedChainClassLoader*. Using the class loader of an application instance and reflection the *SharedChainClassLoader* is able to access over the so-called Application Bridge the simulator bridge of each application individually.

2.2 Event Processing

The aspects woven into the simulated applications transform various API interactions into events which are then dispatched to Mimicry's event engine (cf. Figure 2.2). This dispatching is done by the so-called *Event Bridge* that furthermore manages all blocked control flows of the applications. The generated events are tagged with the application and control flow id and then passed to the underlying event stack. This stack can be configured per node in the simulation script. The event handler are responsible for implementing the actual simulation model you want to apply. Depending on the direction events are passed through the event stack they are called downstream or upstream events. An event handler is allowed to suppress events as well as generating new ones (even asynchronous). Event that reach the bottom of the event stack are dispatched to the event broker that notifies all other nodes as well as further listeners.

2.3 Application Lifecycle

One important functionality of a simulator is to shutdown simulated application instances when desired. And indeed Mimicry is able to shutdown any

application even if they run in infinite loops or are blocked due to synchronization. This is basically realized introducing aspects which cover thread instantiation, synchronization as well as loop interception. At the moment AspectJ is not capable of weaving loops but Harbulot and Gurd already proposed a possible extension in [HG06] and created a prototype implementation LoopsAJ (http://cnc.cs.man.ac.uk/projects/loopsaj/) which is based on top of abc (http://www.sable.mcgill.ca/abc/) and Soot (http://www.sable.mcgill.ca/soot/). However, using the soot framework Mimicry is already intercepting loop headers and checking each iteration whether the current managed thread is in shutdown phase. If so it throws a ThreadShouldTerminateException which is basically the same mechanism the JDK uses when *Thread::destroy* is invoked. Furthermore the user code is prevented from catching this exception accidentally but intercepting all catch clauses and re-throwing that particular type of exception. Finally, all timing related reason for blocking a control flow are handled with the actual clock implementation that works closely together with all threading aspects.

2.4 Cluster Event Processing

In order to allow Mimicry to scale to thousands of simulated nodes and applications the framework provides support for running Mimicry on a cluster of JVMs. This feature is currently not fully implemented.

Chapter 3

Extending the Mimicry Framework

The Mimicry framework is meant to be extended by user simulations. A common case is to write custom event handler that implement special handling of TCP connection, e.g. simulation of bandwidth, jitter models, etc. Therefore this chapter shows how the most common extension points of Mimicry can be used. All extensions can be used without recompiling Mimicry itself. For this purpose several directories are used to create the framework's classpath:

- plugins/

  This directory contains all event handler classes as well as dependencies of themselves. You can also refer to this code from within the simulation script.

- lib/aspects/

  Contains all compiled aspects that are applied to the application code. While this could be used to create new extension points within an application it's not recommended to do so without reading the existing aspects to avoid interferences.

- lib/bridge

  This directory contains all code that is loaded into the address space (class loader) of each application instance. This code is typically referenced by the aspects to generate events.

- lib/shared/

  Contains all classes shared between the bridge and the core framework. You would typically place your custom event classes within it.

- lib/core/

  This directory is not meant to contain custom user code.

3.1 Event Handler

The most common extensions are event handler that are necessary for each simulation. Therefore great care has been taken to create a simple but still powerful as well as robust API. All event handler need to implement the interface *com.gc.mimicry.core.event.EventHandler* and must provide a publicly visible default constructor (since instantiation is done by Mimicry internally when required).

```
1  package com.gc.mimicry.core.event;
2
3  import com.gc.mimicry.core.timing.Clock;
4  import com.gc.mimicry.core.timing.Scheduler;
5  import com.gc.mimicry.shared.events.Event;
6
7  /**
8   * An {@link EventHandler} is part of an {@link EventStack} attached to
        a {@link Node}. It's highly recommended not to
9   * create any threads within an {@link EventHandler}, instead use the
        given {@link Scheduler}. As long as the event
10  * handler is using only the given {@link Scheduler} instance for
        performing asynchronous tasks it has not to consider
11  * any thread synchronisation. By default all methods invoked on this
        event handler are performed in a dedicated thread
12  * (the "Event Handler Thread" - EHT) to this handler and therefore
        thread-safe. This also applies for jobs being
13  * executed by the given scheduler.
14  *
15  * @author Marc-Christian Schulze
16  *
17  */
18 public interface EventHandler
19 {
20     /**
21      * Initializes this handler instance after it has been created and
            before being attached to the {@link EventStack}.
22      *
23      * @param scheduler
24      *            Use this scheduler for all asynchronous operations
            required by this handler. The scheduler will use
25      *            the EHT to run the scheduled jobs which makes each
            event handler fully thread-safe.
26      * @param clock
27      *            A clock to obtain the current time of the simulation.
            Note that this clock is not necessarily
28      *            synchronized with the real-time.
29      */
30     public void init(EventHandlerContext ctx, Scheduler scheduler, Clock
           clock);
31
32     public Scheduler getScheduler();
33
34     /**
35      * Gets invoked when an event is passed down in the {@link
```

```
         EventStack} which means it's an outgoing event of the
36       * application. This method must not block. If you need to delay the
             event forwarding use the {@link Scheduler}
37       * passed in the constructor. To pass the given event further down
             or up you can use the
38       * {@link EventHandlerContext#sendDownstream(Event)} and {@link
             EventHandlerContext#sendUpstream(Event)} methods.
39       * This method is only invoked from within the EHT.
40       *
41       * @param evt
42       *           The event passed downstream.
43       */
44      public void handleDownstream(Event evt);
45
46      /**
47       * Gets invoked when an event is passed up in the {@link EventStack}
             which means it's an incoming event to the
48       * application. This method must not block. If you need to delay the
             event forwarding use the {@link Scheduler}
49       * passed in the constructor. To pass the given event further up or
             down you can use the
50       * {@link EventHandlerContext#sendUpstream(Event)} and {@link
             EventHandlerContext#sendDownstream(Event)} methods.
51       * This method is only invoked from within the EHT.
52       *
53       * @param evt
54       *           The event passed upstream.
55       */
56      public void handleUpstream(Event evt);
57  }
```

The two primary methods are handleUpstream and handleDownstream which are invoked by the event stack when events are passed up or down. You can also use a more abstract base class named *EventHandlerBase*.

```
1   package com.gc.mimicry.core.event;
2
3   import com.gc.mimicry.core.runtime.Application;
4   import com.gc.mimicry.core.timing.Clock;
5   import com.gc.mimicry.core.timing.Scheduler;
6   import com.gc.mimicry.shared.events.Event;
7   import com.google.common.base.Preconditions;
8
9   /**
10   * Base class for most of the {@link EventHandler}s.
11   *
12   * @author Marc-Christian Schulze
13   *
14   */
15  public class EventHandlerBase implements EventHandler
16  {
17      @Override
18      final public void init(EventHandlerContext ctx, Scheduler scheduler,
            Clock clock)
```

```
19      {
20          Preconditions.checkNotNull(ctx);
21          Preconditions.checkNotNull(scheduler);
22          Preconditions.checkNotNull(clock);
23
24          context = ctx;
25          this.scheduler = scheduler;
26          this.clock = clock;
27
28          initHandler();
29      }
30
31      @Override
32      final public Scheduler getScheduler()
33      {
34          return scheduler;
35      }
36
37      final public Clock getClock()
38      {
39          return clock;
40      }
41
42      final public EventHandlerContext getContext()
43      {
44          return context;
45      }
46
47      /**
48       * Override to handle events passed down the {@link EventStack}.
49       *
50       * @param evt
51       */
52      @Override
53      public void handleDownstream(Event evt)
54      {
55          context.sendDownstream(evt);
56      }
57
58      /**
59       * Override to handle events passed up in the {@link EventStack}.
60       *
61       * @param evt
62       */
63      @Override
64      public void handleUpstream(Event evt)
65      {
66          context.sendUpstream(evt);
67      }
68
69      /**
70       * Send the given event to the next event handler upstream in the
             {@link EventStack}. Once the top of the
```

```
71      * {@link EventStack} is reached the event is dispatched to the
            application identified by the id within the
72      * {@link Event#getTargetApplication()} attribute. If no such
            application exists the event is dropped. If you
73      * override this method make sure that you pass all events not of
            your interest upstream. Otherwise you would
74      * suppress the event.
75      *
76      * @param evt
77      *           The event received either from an {@link EventHandler}
            higher in the {@link EventStack} or one of the
78      *           {@link Application} running this {@link Node}.
79      */
80     protected void sendUpstream(Event evt)
81     {
82         context.sendUpstream(evt);
83     }
84
85     /**
86      * Send the given event to the next event handler downstream in the
            {@link EventStack}. Once the bottom of the
87      * {@link EventStack} is reached the event is dispatched using the
            {@link EventBroker} to the event stacks of all
88      * other nodes.If you override this method make sure that you pass
            all events not of your interest downstream.
89      * Otherwise you would suppress the event.
90      *
91      * @param evt
92      *           The event received either from an {@link EventHandler}
            lower in the {@link EventStack} or the
93      *           {@link EventBroker}.
94      */
95     protected void sendDownstream(Event evt)
96     {
97         context.sendDownstream(evt);
98     }
99
100    /**
101     * Override this method to initialize the handler after scheduler
            and clock have been set. This method is invoked
102     * only once per instance.
103     */
104    protected void initHandler()
105    {
106    }
107
108    private Scheduler scheduler;
109    private Clock clock;
110    private EventHandlerContext context;
111 }
```

The use of the base class is recommended if you either are only processing upstream or downstream events; or you don't have to subclass anything else. It's important to note that the event handler are entirely thread-safe as long as you

don't spawn your own thread within. Instead use the given Scheduler instance which is synchronized with all other thread access to your handler instance. Furthermore you shouldn't create any UI elements such as frames or dialogs within your handler because it's not always the case that they are instantiated within your local JVM. Sometimes you want to separate your handling code into different layers like in the ISO OSI model but still access the state of the other handler. Mimicry therefore has a built-in feature to obtain references to event handler within the same event stack. The *EventHandlerContext* provides a method named *findHandler* that takes a class and returns a proxy to the handler instance.

```
1 MyHandler handler = getContext().findHandler(MyHandler.class);
```

The returned proxy can be safely invoked and the access is serialized on the thread responsible for the event handler. Note that obtaining the proxy is quite expensive and should therefore be done in the initialization method. You can create proxies from interfaces which internally uses JDK's dynamic proxies as well as of classes which in that case uses CGLib.

Finally you might want to make your handler configurable by the simulation script. This can easily be achieved by implementing another interface called *Configurable*.

```
1  package com.gc.mimicry.core.runtime;
2
3  import java.util.Map;
4
5  import com.gc.mimicry.core.event.EventHandler;
6  import com.gc.mimicry.core.event.EventStack;
7
8  /**
9   * Implement this interface by your {@link EventHandler} to allow
        configuration using the
10  * {@link EventHandlerConfiguration}.
11  *
12  * @author Marc-Christian Schulze
13  *
14  */
15 public interface Configurable
16 {
17     /**
18      * Invoked after the event handler has been created but before it's
            attached to the {@link EventStack} and before it
19      * gets initialized.
20      *
21      * @param configuration
22      *           The configuration as specified in the {@link
            EventHandlerConfiguration} set up in the simulation
23      *           script.
24      */
25     public void configure(Map<String, String> configuration);
26 }
```

Once you've implemented that interface the framework will automatically inject the configuration provided in the simulation script into your event handler. The definition of the configuration might look like this:

```
EventHandlerConfiguration[] eventStack =
[
  [
    className: "org.example.MyHandler",
    configuration:
    [
      key1: "value1",
      key2: "value2"
    ]
  ],
  ...
]
```

3.2 Event Listener

Event listener allow you to write code that receives events without being part of any event stack. They can directly be registered at the *EventBroker*. Unlike event handler they are not able to suppress events. They can only monitor the event stream and inject events asynchronously.

```
package com.gc.mimicry.core.event;

import com.gc.mimicry.shared.events.Event;

/**
 * Implement this interface to register yourself as listener to the
     {@link EventBroker}.
 *
 * @author Marc-Christian Schulze
 *
 */
public interface EventListener
{
    /**
     * Invoked when an event has occurred.
     *
     * @param evt
     */
    public void handleEvent(Event evt);
}
```

They are typically used to write plug-ins that are not located within an event stack but directly instantiated in the simulation script. An example of such a plug-in is the *ConsoleWindowPlugin* that can be attached to an application in order to interaction with the command line of that particular instance.

```
import com.gc.mimicry.plugin.ConsoleWindowPlugin;
```

```
// ...
ConsoleWindowPlugin.attach(network.getEventBroker(), appRef);
```

3.3 Custom Event Types

All events must implement the *Event* interface in order to processable by the event engine.

```
package com.gc.mimicry.shared.events;

import java.io.Serializable;
import java.util.UUID;

/**
 * Basic interface for all events of the system.
 *
 * @author Marc-Christian Schulze
 *
 */
public interface Event extends Serializable
{
  /**
   * Returns the id of the associated control flow or null if no control
       flow
   * is associated.
   *
   * @return
   */
  public UUID getAssociatedControlFlow();

  /**
   * Returns the id of the application which caused this event or null
       if this
   * event was not caused by an application.
   *
   * @return
   */
  public UUID getSourceApplication();

  /**
   * Returns the id of the application this event is destined for or
       null if
   * not directly destined for a application.
   *
   * @return
   */
  public UUID getTargetApplication();
}
```

In the current implementation the events don't make use of the VectorClock implemented in the Mimicry core which can later on be used to establish happened-before relations among recorded events.

3.4 Built-In Event Types

This section describes the existing event types and how they are raised and consumed by the application.

3.4.1 Console Events

The console of each application instance currently provides the following event types:

ConsoleInputEvent
Can be emitted either by the simulation script or a plugin, e.g. the ConsoleWindowPlugin does exactly this when entering some text in the window.

ConsoleOutputEvent
Those events are emitted by the applications when it is writing something to stdout or stderr. Currently both streams are aggregated in the same event and no differentiation is possible.

3.4.2 Networking Events

The networking aspects currently provide the following event types:

SocketBindRequestEvent
Is emitted by *java.net.Socket*s that try to bind to a certain port. This event typically has a control flow associated that is blocked. You can respond to this event either with a SocketClosedEvent or a SocketBoundEvent.

SocketClosedEvent
Is either raised by the application when the *java.net.Socket* has been closed or by an event handler when he decides to asynchronously close the socket. This event type does not require a control flow to be set. It's automatically picked up using the endpoint address.

SocketConnectionRequest
Raised by the application when a *java.net.Socket* tries to connect to a certain address. You can either respond with a SocketClosedEvent or a ConnectionEstablishedEvent.

ConnectionEstablishedEvent
Raised by the event handler to indicate that a TCP/IP connection was successfully established.

SetSocketOptionEvent
Emitted by the application when a socket option has been changed.

SetPerformancePreferencesEvent
Raised by the application when the QoS parameters of the socket have been changed.

TCPSendDataEvent
Emitted by the application when someone writes something into the OutputStream associated with the socket.

TCPReceivedDataEvent
Emitted by the event handler to store data into the receive buffer of a TCP/IP socket which is picked up by the application using the socket's associated InputStream.

SocketAwaitingConnectionEvent
Emitted by a ServerSocket when its accept method is invoked by an application. Can be either responded to by a SocketClosedEvent or a ConnectionEstablishedEvent.

SetDatagramSocketOptionEvent
Emitted by DatagramSockets when the socket options are changed.

SetMulticastSocketOptionEvent
Emitted by MulticastSockets when the socket options are changed. Note that the common options with a DatagramSockets are indicated by a SetDatagramSocketOptionEvent.

JoinMulticastGroupEvent
Emitted by MulticastSockets if the joinGroup method is invoked.

LeaveMulticastGroupEvent
Emitted by MulticastSockets if the leaveGroup method is invoked.

UDPPacketEvent
Either emitted by a DatagramSocket, a MulticastSocket or an event handler.

Bibliography

[HG06] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 63–74, New York, NY, USA, 2006. ACM.