1. Introduction

1.1. Introduction

SemiSpace is a tuple space oriented towards being easy to integrate with existing technology. The module semispace-main is the only module that you really need to have a dependency to, and the libraries that this module depends on, are intended to be few.

This document will explain different configuration and usage strategies, and present a tutorial which is a practical guide.

SemiSpace contains many modules and usage models. Try not to let that confuse you; Only semispace-main is mandatory, the other modules are used on a need to have basis: If you do not need the web services interface, disregard the module. If you do not want to use, or need, Spring, use standard POJO code for your configuration.

1.1.1. Differences from traditional implementations.

Wikipedia gives an in depth discussion of what a Tuple Space is. SemiSpace's has a focus that is slightly different from the the traditional implementations as the space itself is not perceived to be separate from the client(s) VM(s). How you use it, is analogous, though. When having more than one VM, Terracotta takes care of the distribution.

SemiSpace does not use the concept of client and server. Rather, the space is distributed over the nodes present in the space.

Getter objects are supported as well as public fields. The JavaSpace standard only lets you play with public fields, which is a nuisance if you want to use (possibly) auto-generated Hibernate objects.

As long as your class can be streamed with XStream, you can put it into SemiSpace.

1.1.1.1. SemiSpace does not use Jini

SemiSpace does not require installation of a Jini server.

Jini is the crux of the standard JavaSpaces - which are the ones that implement the JavaSpace interface. Jini uses JERI, which is a re-implementation of RMI. JERI supports dynamic stub

generation, which is useful if you are adhering to the client / server idiom.

The methods you find in the SemiSpace interface, however, are analogous to the methods you find in the JavaSpace interface. Presently JavaSpace05 is not considered. The differences between the SemiSpace and JavaSpace interface, is mainly that the transaction object has been removed, and that the return types may be different.

1.1.1.2. Terracotta is used for distribution

SemiSpace is bundled with a [Terracotta](#) Integration Module ([TIM](#)), which you can use in order to distribute the contents of your space. Besides distribution, you have the benefit of not having to create the configuration yourself.

1.1.1.3. SemiSpace uses XML

The transport and internal storage layer of SemiSpace is XML. This has the disadvantage of bloating and transformation, but the advantage of not needing any dependencies on Java objects when treating and using the space. This means that you can use the same space for several purposes without worrying about getting ClassCastExceptions.

When you are using Java code directly on the space, the XML layer is hidden, and you can concentrate on using the space in a manner which is the same as if you would use a [Jini](#) based implementation.

1.1.1.4. Query on first level depth

SemiSpace allows a nested object structure, but queries are only performed on the first level. This has the advantage have rather fat holder objects, whilst retaining the speed and functionality of the space. This make the space usable for caching.

This implies, however, that you should model the holder objects of the space carefully, preparing them with structures that contain record-like data, such as String and Integer. Large byte arrays, for instance, on the first level is discouraged as it becomes one of the fields that are used for identification - and as such is propagated over your servers.

1.1.1.5. All this implies:

- You can interface the space from different languages (such as [Ruby](#)) without having to think about Java Objects. The interfacing is performed with web services, in this case, and you will

need to set up the web service end point.
- Java implementation on different nodes do not need to have the same libraries available.
- Use of SemiSpace is easy to junit test, as the space works well on a single VM.
- When distributing with Terracotta, you do not be too concerned about tc-config.xml (the Terracotta configuration file).
- You can distribute Objects without taking the object depth into consideration. In other words, can you have arbitrarily large and / or nested objects. Only the first level is used in matching, though, and you still need to pay heed to network traffic overhead.

1.2. Do not distribute for its own sake

Much of the traditional view of tuple spaces concentrate on distribution of workload. Whereas this can be a benefit, I often find it overrated. In order to benefit from distribution, you need an problem which is scalable in itself. Amdahl's law explains that the speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

Additionally, you have the question of network pipe. Let's say you want to distribute image scaling. With images having a "normal" compact camera size, you quickly begin to test the network instead. On the other hand, you may remove load from the from end servers.

Everything has to be done for a reason, and figuring those reasons out in advance has it's benefit.

1.2.1. Not everything is a nail

It might be tempting to use tuple oriented problem solving on everything that needs distribution. For instance, if you need a distributed file system and distributed processing, Hadoop with its Map Reduce implementation might be more suitable. The converse is also true, of course. Map reduce has its benefits, but is not suitable for all problems.

1.3. Reasons for using SemiSpace

Even though you can use SemiSpace on a standalone server, you benefit the most when you distribute.

Distribution idioms are easy to implement and maintain, whether this is used for caching, information gathering or maintenance of session information.

1.3.1. Failover and load balancing

A fairly standard setup, is to have two or more front end servers accepting queries in a load balanced manner. SemiSpace can be set up in order to make the the impact of adding a new node minimal.

When having back end servers treating queries and data, you can add specialized processing machines for treating workloads that are large. Consider having 5 nodes, and figuring out that graph generation takes a lot of time. Re-assign one of the nodes, or add a new one, which is dedicated to only graph generation. This can be quite simple if the task is not too interconnected with the rest of the application.

1.3.2. Time bound processing

SemiSpace is similar to JavaSpace in that processing can be time bound. This has the benefit that you can guarantee an answer (as no answer also is an answer). Lets say you have 10 elements that need to be processed for you web page, and only the content retrieval is critical. The other parts may be graphs, statistics or other non-critical information. You can setup your application to await the content bulk, but disregard the rest if it has not finished processing.

Time bound processing is also useful when dealing with animations and other data that need to be supplied in a tick-tock manner, i.e. regularly. You need to move the animation even if not all the data is present.

1.3.3. Avoiding spiraling death

Let's say you have 5 front end servers. One of them gets a problem due to workload and goes down. If this makes the other 4 servers fail, you have a spiraling death problem, and in effect, you may need to take all servers out of commission while you correct the problem. (This actually is a lot more common than you think.)

That servers bounce up and down is normal. The impact on the overall service should be minimized. This can be done by having (that is, changing an existing) architecture to degrade gracefully.

Lets say you have two components, content generation and graph generation. Content is critical, and graph generation is not - even if it generates the most load. Let both generate their content in a master / worker manner. Give content generation 10 workers and graph generation only 2 (multiply with factors as you see fit). The timeout for waiting for a graph should be reasonable. As the graph generation now is load bound, one missing server does not take the other servers down.

1.3.4. Inherent caching

If you are distributing queries, you may get an out-of-the box cache by simply first querying to find if an equal query has already given a result. This is performed by not performing a take on the space, but rather a read.

1.3.5. Divide and conquer your program

Even when you are not having load or distribution problems, you will benefit from dividing your program into separate autonomous parts. This makes it easier for multiple programmers to divide tasks between themselves.

2. Installation

2.1. Preliminaries

SemiSpace runs on J2SE 6.0 or greater. The other components are optional. If you are not intending to distribute, you will not need Terracotta, etc.

If you want SemiSpace to be distributed, you need to download and install Terracotta.

In order to benefit from the webservices proxy (more about this in the tutorial), you need a sensible endpoint. This is in effect a webapp. Choose this at your own discretion. The recommendation is Jetty , but you should be effortlessly able to use Tomcat , or Geronimo as well.

It is presumed that you use bash as shell, most likely under MacOSX or Linux. Scripts and instructions must be adapted at your discretion if you are running something else.

2.1.1. XStream support jars

XStream has some optional dependencies, and the XML support library xpp has been configured to be default, but you can exclude it and include a different support library instead. See http://xstream.codehaus.org/download.html for details, or just include the following in your maven dependencies:

```
<dependency>
   <groupId>xpp3</groupId>
   <artifactId>xpp3_min</artifactId>
   <version>1.1.4c</version>
</dependency>
```

2.1.2. Sun jars

The persistence and webapp sub projects depend on sun jars, which you may have to download and install yourself.

See Maven's guide to coping with Sun JARs for more information.

The semispace-main project does not have any such dependencies, indeed it is depends only on XStream and slf4j logging.

This chapter can be skipped, if you are not going to use Terracotta, or if you are not going to use Terracotta right away.

This explanation is aimed at Terracotta version 3.4.1.

2.2. Configuration of Terracotta

You can use SemiSpace's tc-config as a starting point for your configuration.

You can download the configuration SemiSpace is set up to use from the subversion repository:
http://www.semispace.org/svn/trunk/tc-config.xml
The TIM version can be found here:
http://www.semispace.org/svn/trunk/tc-tim.xml

The configurations can be modified to your own needs. You find comments inside it which should help you along. See Terracotta documentation for more information about configuration.

You want to set the environment variable TC_INSTALL_DIR pointing at the, surprise, installation directory of Terracotta:

```
export TC_INSTALL_DIR=<where you have exploded the terracotta-3.x.x.tar.gz file>
```

2.2.1. Using Maven for SemiSpace TIM reference

The easiest way of configuring SemiSpace for Terracotta, is to use Maven and add a <repository/>section in your tc-config.xml. This tells Terracotta where it can find these modules:

```
<modules>
    <!-- You only need jetty module if you are actually using jetty -->
    <module name="tim-jetty-6.1" version="2.2.1"/>
    <repository>%(user.home)/.m2/repository</repository>
    <module name="semispace-tim" version="1.3.0" group-id="org.semispace"/>
</modules>
```

Notice that the bundle reference only works with released versions, as it does not manage to parse "SNAPSHOT" as an integer.

--- assuming that your local Maven repository is at:

```
~/.m2/repository
```

2.2.2. Jetty TIM module

If you are going to use Jetty, you need the jetty TIM module for the Terracotta configuration, if this has not already been installed. You can check and obtain it by doing as follows:

```
$TC_INSTALL_DIR/bin/tim-get.sh list jetty
$TC_INSTALL_DIR/bin/tim-get.sh install tim-jetty-6.1 2.2.1 org.terracotta.modules
```

2.2.3. Starting the server

Start the terracotta server with an invocation similar to:

```
${TC_INSTALL_DIR}/bin/start-tc-server.sh -f some/path/to/config/tc-config.xml
```

The server part of the configuration file may look something like this:

```
  <servers>

    <!-- When having remote clients in a multiserver setup, you _must_ give the host an IP
address -->
    <server host="%i" name="localhost">
      <dso-port>9510</dso-port>
      <jmx-port>9520</jmx-port>
      <l2-group-port>9530</l2-group-port>
      <data>/tmp/terracotta/server-data</data>
      <logs>/tmp/terracotta/server-logs</logs>
      <!-- If you are sharing disk space over nfs or NAS, or are running in single VM modus
      <dso>
        <persistence>
          <mode>permanent-store</mode>
        </persistence>
      </dso>
       -->
      <dso>
        <persistence>
```

```xml
        <mode>temporary-swap-only</mode>
      </persistence>
    </dso>
  </server>
  <!-- If you have some other server available
  <server host="servername.some.domain" name="servername">
    <dso-port>9510</dso-port>
    <jmx-port>9520</jmx-port>
    <l2-group-port>9530</l2-group-port>
    <data>/tmp/terracotta/server-data</data>
    <logs>/tmp/terracotta/server-logs</logs>
    <dso>
      <persistence>
        <mode>temporary-swap-only</mode>
      </persistence>
    </dso>
  </server>
   -->
  <ha>
    <mode>networked-active-passive</mode>
    <networked-active-passive>
      <election-time>5</election-time>
    </networked-active-passive>
  </ha>
 </servers>
 <clients>
  <logs>terracotta/client-logs</logs>
  <!-- Load jetty-module. Notice that Eclipse integration will fail
     if TIM support has not been installed. The module version
      may exist in different versions-->
<modules>
  <!-- module name="tim-jetty-6.1" version="2.2.1"/ -->
     <!-- module name="semispace-tim" version="1.3.0" group-id="org.semispace"/ -->
</modules>
 </clients>
```

Notice that you need to exchange the "%i" with a real DNS name or IP address if you are using remote clients that receive the configuration from the server. Otherwise they will simply not find the server...

2.2.4. Sharing the configuration with clients

Clients need to use the same configuration as the Terracotta server. This is most easily performed by letting the clients retrieve the configuration from the server (exchanging localhost with the name of your real host if applicable):

```
${TC_INSTALL_DIR}/bin/dso-java.sh -Dtc.config=localhost:9510
```

In addition to the configuration of the Terracotta server itself, you need to either configure up SemiSpace to use java objects, or to retrieve the space configuration from spring.

The Terracotta configuration can naturally be amended with your "other" configuration as well; Just add it. This may be useful if you also want to, lets say, distribute the webapp session objects.

Notice that only some configuration highlights are presented here. You will benefit from reading more about Terracotta on their home page.

2.2.5. POJO configuration

This is typically the configuration you will use outside a webapp. Note that this is not relevant if you use the SemiSpace TIM, as the TIM will take care of the configuration for you.

If you prefer to obtain a distributed reference to Terracotta without using Spring, you need to add the relevant SemiSpace specific class files.

The relevant configuration snippet for this is:

```
<application>
 <dso>
   <!-- Could not share a common root between different
      contexts even when https://jira.terracotta.org/jira/browse/CDV-272
      indicates I can:
   <app-groups>
    <app-group name="space">
     <web-application>semispace-war</web-application>
```

```xml
    <web-application>semispace-google</web-application>
  </app-group>
</app-groups>
  -->
<instrumented-classes>
  <include>
    <class-expression>org.semispace.HolderContainer</class-expression>
    <honor-transient>true</honor-transient>
  </include>
  <include>
    <class-expression>org.semispace.Holder</class-expression>
    <honor-transient>true</honor-transient>
  </include>
  <include>
    <class-expression>org.semispace.HolderContainer</class-expression>
    <honor-transient>true</honor-transient>
  </include>
  <include>
    <class-expression>org.semispace.HolderElement</class-expression>
    <honor-transient>true</honor-transient>
  </include>
  <include>
    <class-expression>org.semispace.EventDistributor</class-expression>
    <honor-transient>true</honor-transient>
  </include>
  <include>
    <class-expression>org.semispace.SemiSpaceStatistics</class-expression>
    <honor-transient>true</honor-transient>
  </include>
    <include>
      <class-expression>org.semispace.event.SemiEvent</class-expression>
    </include>
    <include>
      <class-expression>org.semispace.event.SemiExpirationEvent</class-expression>
    </include>
  <include>
    <class-expression>org.semispace.event.SemiAvailabilityEvent</class-expression>
  </include>
  <include>
    <class-expression>org.semispace.event.SemiTakenEvent</class-expression>
  </include>
  <include>
    <class-expression>org.semispace.event.SemiRenewalEvent</class-expression>
  </include>
</instrumented-classes>
<locks>
```

```xml
    <autolock>
      <method-expression>* org.semispace.HolderContainer.removeHolderById(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.removeEmptyHeads(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.addHolder(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.findById(long,String)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.size(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.retrieveGroupNames(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.waitHolder(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.readHolderWithId(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderContainer.retrieveClassNames(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>

    <autolock>
```

```xml
      <method-expression>* org.semispace.HolderElement.size(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderElement.removeHolderById(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderElement.addHolder(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderElement.toArray(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderElement.iterator(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderElement.isWaiting(..)</method-expression>
      <lock-level>read</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.HolderElement.waitHolder(..)</method-expression>
      <lock-level>write</lock-level>
    </autolock>

    <autolock>
      <method-expression>* org.semispace.Holder.setLiveUntil(long)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
    <autolock>
      <method-expression>* org.semispace.EventDistributor.distributeEvent(DistributedEvent)</method-expression>
      <lock-level>write</lock-level>
    </autolock>
  </locks>
  <roots>
    <root>
      <field-name>org.semispace.HolderContainer.instance</field-name>
    </root>
  </roots>
```

```xml
    <!-- transient-fields>
      <field-name>org.semispace.SemiSpace.listeners</field-name>
      <field-name>org.semispace.SemiSpace.admin</field-name>
      <field-name>org.semispace.SemiSpace.xStream</field-name>
    </transient-fields -->
    <!-- distributed-methods>
      <method-expression>void
org.semispace.SemiSpace.notifyListeners(org.semispace.EventDistributor)</method-
expression>
    </distributed-methods  -->
    <distributed-methods>
      <method-expression>void
org.semispace.EventDistributor.distributeEvent(org.semispace.DistributedEvent)</method-
expression>
    </distributed-methods>
  </dso>
```

2.2.6. Spring configuration

If you are running within a webapp, and are using spring, you can add the following. Note that this is not relevant if you use the SemiSpace TIM. You will need a named configuration element for each of your webapps. Depending on your scenario, you may find it easier / more maintainable to just define and use the application in a POJO manner.

```
<!--
  Clustering Spring no longer requires special configuration. For more information, see
http://www.terracotta.org/spring.
  <spring>
    <jee-application name="*">
      <application-contexts>
        <application-context>
          <paths>
            <path>*.xml</path>
          </paths>
          <beans>
            The other beans to share beside semispace
            SemiSpace is covered in the dso configuration, and we do NOT need: <bean
name="semispace" />
          </beans>
        </application-context>
      </application-contexts>
    </jee-application>
  </spring>-->
```

2.3. You can only mix some Terracotta runtime configurations

Mixing runtime configurations can give you some hassle. Terracotta lives best when running one of the following:

- Running with a POJO container, such as a standalone console program.
- Running Spring configured in a webapp.
- Running POJO-invoked in a webapp.

It is possible to mix some configurations, but it may involve an effort. The reason and explanation can be found here: Object identity in Terracotta is dependent on the classLoaderName + fully qualified reference name .

2.3.1. Mixing console app and webapp

Presuming you are running with Jetty, the following extra parameter will interface your console application with your webapp:

```
-Dcom.tc.loader.system.name="Jetty.path:/some_path"
```

Terracottas documentation explains this in greater detail. The problem is even larger if you are using spring to configure the application.

2.4. Running with Terracotta within Eclipse

For test purposes, or ease of development, you may want to run Terracotta within Eclipse.

Besides Eclipse itself, you will need the Terracotta eclipse plugin . Activate your project as a "Terracotta DSO project".

If you do not have a tc-config.xml file already, create one as explained above.

Presumably, your program is already using SemiSpace. Then the easiest way of testing the Eclipse / Terracotta easiest way of performing tests, is to create a junit test . Then choose to run the tests as "Terracotta DSO junit test". This will prompt you to start a local Terracotta server, which you may have to do, depending on the contents of you tc-config.xml file.

Notice that you probably want to remove the TIM module from the configuration whilst running under Eclipse, as this will make the integration fail. Presumably, you do not need Jetty integration in Eclipse.

3. Tutorials

3.1. Introduction to tutorials

The tutorials will try to introduce different aspects of SemiSpace, and how to use it.

You can decide yourself whether or not you want to download any sources. The tutorial is intended to be self contained, and the examples should be possible to followed by instruction alone.

3.1.1. Downloading the source code (optional)

If you want to follow the source more closely than the snippets offer, you can download the source code by checking it out with subversion from http://www.semispace.org/svn/ Choose a given tag or trunk.

You can also check out individual project parts with your favorite IDE.

3.1.1.1. Building the source code

Building the source code is quite straight forward with Maven2:

```
mvn clean install
```

You will need version 2.2.1, or later, of maven.

If you get a dependency error due to missing JAXB, you need to install JAXB as explained in the preliminaries chapter.

You can skip the test by using the dev profile:

```
mvn -Denv=dev clean install
# or in the regular maven way:
mvn -Dmaven.test.skip=true
```

Notice that the essential project is semispace-main - the rest is for your convenience. If you have problems compiling (typically because of missing jar libraries), you can try to comment out optional modules in the parent pom.

3.1.1.2. Failing comet tests

The comet tests need a running comet instance. This is most easily obtained by

- compiling the complete project without any tests
- check out semispace-comet in a separate directory
- run the comet server with

```
mvn jetty:run
```

- run the tests in the source version that contain the complete project.

3.1.2. Terracotta for distribution

Many of the examples will use terracotta for distribution.

The following environmental variables are used in the tutorials:

```
export TC_INSTALL_DIR=/some/path/to/terracotta-3.X/

export TC_CONFIG_PATH="localhost:9510"
# or
export TC_CONFIG_PATH=/some/path/to/tc-config.xml
```

You use TC_CONFIG_PATH="localhost:9510"if you want the Terracotta server to give you the configuration during startup.

3.2. Stand alone operation

In order to use the space in a stand alone fashion in your java code, just obtain a reference to it, and use it. You will need to use threading in order to benefit from it, as you typically will have an application thread and a worker thread:

```
SemiSpaceInterface space = SemiSpace.retrieveSpace();
```

3.3. General space usage

In order to insert an element into the space, use write:

```
Element element = new Element();
element.setName(args[0]);
element.setValue(args[1]);
SemiSpaceInterface space = SemiSpace.retrieveSpace();
// Life time of 5 minutes.
space.write( element, 1000*5*60);
System.out.println("Element inserted successfully:
"+element.getName()+"="+element.getValue());
```

Similarly, in order to read an element in the space, use:

```
Element searchFor = new Element();
searchFor.setName(args[0]);
SemiSpaceInterface space = SemiSpace.retrieveSpace();
// reading with a timeout of 60 seconds
Element read = space.read(searchFor, 60000);
if ( read == null ) {
    System.out.println("Could not find an element with name "+searchFor.getName());
} else {
    System.out.println("Element found: "+read.getName()+"="+read.getValue());
}
```

In order to remove an element from the space do:

```
Element read = space.take(searchFor, 60000);
```

The essential difference, is the read statement. Read can be performed repeatedly, whereas take will remove the object from the space.

3.4. Using notify

Notify gives you an event containing the XML source of the element which matched the registration.

A simple example of this is the following. First you need to register a notification:

```
        SemiSpaceInterface space = SemiSpace.retrieveSpace();
        SemiEventRegistration eventRegistration = space.notify(new Element(), this, 60 *
 1000 * 60);
 // If this comment is seen in the doc, it is because mavens apt book generator has
 // become confused with the code snippets.

        for ( int i=0 ; i < 10 ; i++) {
            try {
                Thread.sleep(1000 * 10);
            } catch (InterruptedException e) {
                // Ignore
            }
        }
        /* If you like to cancel the notification, perform
           the following: */
        eventRegistration.getLease().cancel();
      } catch (RuntimeException e) {
        e.printStackTrace();
      }

    }

}
```

In the notification method itself, you perform whatever you want in case of notification. The following just prints a statement that explains that an object matching the template is found in the space.

```
 public void notify(SemiEvent theEvent) {
    if ( theEvent instanceof SemiAvailabilityEvent) {
       System.out.println("Incoming element which concurs with template has arrived.");
       Element element = (Element) SemiSpace.retrieveSpace().takeIfExists( new Element());
       if ( element ==null ) {
          System.out.println("Could not take element that was flagged as available");
       } else {
          System.out.println("Read element from space:
 "+element.getName()+"="+element.getValue());
       }
    }
 }
```

Sometimes, you will try to take the Object that has been transported with the notification. Be aware that the object may already have disappeared (for instance if it has already been taken). Therefore you always need to test your taken object for null.

3.4.1. Notify and disappearing instance.

You may think that a long living notification may be a problem if the instance falls down when being distributed with Terracotta. This is not the case. The notification lives only within the server instance in question, and if it disappears, it does not matter - the notification just disappears with it. However, the statistics will become wrong - in the count of number of listeners, as the listener is not de-registered correctly. This is inconsequential.

If you like to cancel the registration, you can do this on the registration lease.

3.5. Simple terracotta interaction

The purpose of the tutorial is to give you an insight in how SemiSpace is intended to work together with Terracotta. Terracotta needs to be configured as explained in the installation chapter.

3.6. Running the sources

Presumably, you have the sources downloaded and and installed. Perform:

```
cd semispace-tutorials/semispace-tutorial
mvn clean install
mvn assembly:assembly
cd target
unzip tutorial.zip
chmod a+x *.sh
```

Start the Terracotta server as explained in the Terracotta chapter. Run the java program in order to insert a value into the space:

```
./insert.sh something value
```

You should see a message at the bottom similar to:

```
Element inserted successfully: something=value
```

Then try to retieve it:

```
./take.sh something
```

This should render:

```
Element found: something=value
```

Try to run the take.sh script again. This should not find another instance, and the output should be (after a waiting time, due to 1 minute timeout in wait):

```
Could not find an element with name something
```

If you get any errors, it is likely that you have not configured the environment correct, i.e. forgotten the TC_INSTALL_DIR variable.

The take.sh script performs a take instead of a read, the difference being that the element is removed from the space.

```
Trying notification
```

This program only works when distributed with Terracotta. This is as notification is not exposed over webservices. Furthermore, as this program hangs in an endless loop, you must press CTRL-C to stop it.

The program will use notify, and subscribe to all objects of type Element. This means that you get an output every time you perform an insert into the space. Start the program with notify.sh.

When using the notify.sh , the insert.sh program may report an error. If it does, it is just because it is shutting down at the moment the event is being distributed.

3.7. What can you use this to do

This simple example lets you experiment with the Terracotta configuration. Let's say you want to test Terracotta failover. Perform the following steps:
- Modify the Terracotta "servers" configuration to include two different servers.
- Start Terracotta on both servers
- Run an "insert" (with insert.sh)
- Stop one of the Terracotta instances
- If applicable, modify your TC_CONFIG_PATH value
- Run a "take": Expect everything to still work.
- If you are running the "notify" program, you should get an output for all elements added.

3.8. Case study: Google maps

The purpose of this tutorial is to illustrate SemiSpace usage and space interaction, by making a server able to query and use Google maps API . In order to query Google, you need have a Google maps key. You can still follow the example, even if you do not have a key yet; You can see the interaction in the logs.

The tutorial explains how you can set up a server that takes care of the google communication. It also demonstrates how you can limit which clients that are allowed to use the server.

The swing client also demonstrates how you can use the actor pattern for internal communication.

None of the examples are particularly refined, but they should offer a starting point of understanding the SemiSpace technology, and what you can do with it.

3.9. The webapp server

The purpose of the webapp server, is to create the a control application for google map search. Since all searches are performed on a single server, you can:
- Keep your google application key private
- Cache content lookup
- Refine existing search result by modifying the cache
- Control which clients that are allowed to connect and use the service
- Control to content lookup rate, for instance disallow more than 1 lookup per second

3.10. The swing client

The swing client offers a simple interface, which allows you to log in onto the webapp server. Once logged in, you can perform address lookup queries that are performed on the server. The

communication is performed over web services.

The swing setup is performed with [SwiXML,](#) which is a sensible and pragmatic [XUL](#) framework.

3.11. Starting the webapp and client

The following explains startup of a standalone (development) server without terracotta. Presumably you have, in the top level directory, successfully run

```
mvn clean install
```

3.11.1. Starting the webapp

In order to start the webapp, either put the war file in a suitable webapp container, or run the following:

```
cd semispace-tutorials/semispace-google-webapp
mvn jetty:run
```

When you look at http://localhost:8080/semispace-google/ you should see an entry page. This page allows you to enter users and google key. Notice the link to the WSDL file: http://localhost:8080/semispace-google/services/tokenspace?wsdl This is the endpoint when connecting to the webapp from the standalone application.

3.11.1.1. Alternative to mvn jetty:run

A standalone application has been created as part of the build. Take a peek in the [Jetty / Terracotta integration chapter](#) for how to run it.

3.11.2. Starting the client

Enter the directory where the client reside, and find the target directory:

```
cd semispace-tutorials/semispace-google-client/target
```

You should find a zip file in this directory, which can be used in the following manner:

```
mkdir some_directory
cd some_directory
unzip ../google-client.zip
chmod a+x gclient.sh
```

If you do not run a unix-based operating system, you need to create your own startup script.

Start the client with:

```
./gclient.sh http://localhost:8080/semispace-google/services/tokenspace
```

This shall give you the login page.

3.11.3. Client / server interaction

Without having registered neither users nor google key in the webapp, try enter a (spurious) name and password in the login box, and press login. Notice the log messages.

Now, register a name and password in the webapp. Then try to log in again with the client. The screen should change to allow search expressions.

Enter an address to search - the same way you would do it in Google maps for and press submit. Try, for example,

```
Kongensgate 14, Oslo, Norway
```

The server should log the query, but as no google map key has yet been registered, you naturally do not get a search result. On the client, the query times out because not answer is received.

Register a key and try to search again. You should see the search expression in the server log, and the client should display the information that was retrieved from google.

Notice that if you try to search again with the same key, you receive the cached version from the server.

Try and remove the user by registering a user with the same name as an existing user, but with an empty password. When you try to search in the application, the login box appears again, as the

user is no longer authenticated.

If you do not wish to communicate over webservices, the alternative is to connect up your space with terracotta.

There are many was to do this, depending on how you designed your application. The way that is chosen here, is to use a webapp container - simply because the lookup application is designed as a webapp.

3.12. Terracotta

You need Terracotta with the the jetty TIM module. This has been explained in the installation chapter.

3.13. Jetty

For your convenience, a bundled version of jetty has been created. You could, of course, choose a different app server. But these instructions are for jetty.

If you like to set up your own jetty server and instance, you find instructions here: Clustering Web Applications.

3.13.1. Unpacking jetty application

For your convenience, a bundled jetty application has been created in semispace-google-app . Unpack the zip file and run the preliminary installation script:

```
mkdir somewhere
cd somewhere
unzip <wereever>/google-app.zip
cd bin
chmod a+x afterInstallation.sh
./afterInstallation.sh
```

Notice that you get informed to set some environment variables. These are needed for Jetty to run. The bundled Jetty script is the same script that follows the standard distribution, with the addition of Terracotta specific variables, as explained below.

If you do not add the Terracotta variables, the Jetty instance runs as a standalone server.

3.13.2. Environment variables

You need the following environment variables for connecting to the Terracotta server:

```
export TC_INSTALL_DIR=<path_to_local_Terracotta_home>
export TC_CONFIG_PATH="localhost:9510"
```

These variables are needed for all instances that shall communicate over Terracotta.

3.14. Distributing jetty instances

You need to have unzipped the google-app.zip in two different directories, lets call them A and B. You also need to have started the Terracotta server, as explained in Terracotta configuration chapter.

Recap:

```
export TC_INSTALL_DIR=<where terracotta is installed>
${TC_INSTALL_DIR}/bin/start-tc-server.sh -f some/path/to/config/tc-config.xml
```

3.14.1. Starting up the first jetty instance

It is presumed that you already have configured the paths as applicable to your environment, i.e. exported the variables that are printed after running afterInstallation.sh and exporting TC_INSTALL_DIR and TC_CONFIG_PATH .

Run in the jetty bin directory:

```
./jetty.sh start
```

You should see something similar to:

```
Using Terracotta
Starting BootJarTool...
2009-06-23 13:09:30,165 INFO - Terracotta 3.2.0, as of ....
```

3.14.2. Starting up the second jetty instance

Unzip google-app.zip into a different directory. You need to use a different port as we are running the service on the same machine. If you use two different machines, this, naturally, does not apply. (However, your configuration references would need to be tailored to support this.)

Change the jetty port in etc/jetty.xml from

```
<Set name="port"><SystemProperty name="jetty.port" default="8080"/></Set>
```

to

```
<Set name="port"><SystemProperty name="jetty.port" default="8081"/></Set>
```

3.15. Testing the application:

Open two browser windows: http://localhost:8080/semispace-google/index.html and http://localhost:8081/semispace-google/index.html

This represents your two servers A and B, and should present the same entry page.

Submit a new user in one of the windows. You see the user list is updated with the user. Now, press the index button in the other window. You shall see the same user in that window.

3.15.1. Using the client

You can use the client application over webservices on either servers. The respective endpoints would be: http://localhost:8080/semispace-google/services/tokenspace and http://localhost:8081/semispace-google/services/tokenspace

Example:

```
./gclient.sh http://localhost:8080/semispace-google/services/tokenspace
```

3.15.1.1. Integrating directly from the client

Direct integration from the client is as of Terracotta version 3.x not possible. The problem is that the client does not use spring for setting up the SemiSpace connection, whereas the webapp does, and this does not mix well.

3.15.2. Stopping the server

In the Jetty bin directory do:

```
./jetty.sh stop
```

4. SemiSpace and Cometd

SemiSpace has a module which allows JavaScript to communicate to an cometd -enabled webserver. The javascript interface mimics the Java interface as closely as possible.

We are presently using Cometd-2.x.

4.1. Overview of semispace-comet

The semispace-comet module of SemiSpace consists of the following parts:

semispace-comet-server	The webapp which will answer the JavaScript queries
semispace-comet-client	An emulation of the JavaScript client behaviour. Can be used for emulating a client for test purposes, or for bridging two server implementations
semispace-comet-common	Transport objects and functionality shared between the client and server implementation
semispace-comet-webapp	An example webapp, which gives you inspiration of how to configure your own
semispace-comet-app	A standalone jetty server which you can easily run on the command line
JavaScript	The web client functionality resides in JavaScript which you simply downloaded and add to your webapp

4.2. Installation

When using semispace-comet, you need to install / prepare the server side and the JavaScript side. On the server side is a matter of setting up the webapp with dependencies in a manner

similar to semispace-comet-webapp (which indeed is an example). On the client side, you need to copy down the JavaScript files, and put them in the correct directories.

A good starting point is to examine the projects in semispace-comet, and semispace-comet-webapp in particular. That sub project contains examples of use.

4.3. Instructions for use: TBA

We are on the way of consolidating and organizing the examples in a manner that will suit a presentation like this better. However, we have not got around to do it yet. Sorry.

4.3.1. Dojo binding

TBA

4.3.2. JQuery binding

TBA

Technical overview over channels and parameters.

4.4. Communication channel overview

The communication between the comet client and java webapp comet server is asynchronous. The communication is initiated with a call, and returned with a reply (except in the case of notify, which is a special case).

The communication channels are as follows:

Channel	purpose
/semispace/call/read/ number	Initiate a read request.
/semispace/reply/read/ number	Reply of a read, with the payload being the result if it was obtained.

/semispace/call/take/ number	Initiate a take request.
/semispace/reply/take/ number	Result of take.
/semispace/call/write/ number	Insert an object into the space. This is a synchronous operation when using the comet java client.
/semispace/reply/write/ number	Acknowledge that the element has been written.
/semispace/call/notify/ number / type	Register a notification. This is a synchronous operation when using the comet java client.
/semispace/reply/notify/ number / type	Acknowledge that the notification has been registered.
/semispace/event/notify/ number / type	Attached is a notification event.
/semispace/call/leasecancel/ number	Cancel lease with callId as mapped parameter. CallId is the number from the notify method
/semispace/reply/leasecancel/ number	Acknowledge of listener cancellation

Channel parameters are:

number

       Channel number. When having, for example, two simultaneous read operations, this number must be different for the two operation. The easiest is to have a client side sequence number, which is just incremented for each call.

Notification type

       The type is one of availability , expiration , taken , renewal and all . The Java client only uses all , and the proxy will translate this into the correct response object. This is as the Java client does not know in beforehand what kind of notification that is registered.

It does not matter if several different clients use the same channel numbers. The communication to the client is based on the client ID, and has nothing to do with the channel number.

4.5. Parameter overview

The parameters sent over the channel are packed JSON style. Then the control elements are extracted. The following is an overview:

Operation	Parameter	Significance
read	duration json	How long to wait for an object if it is not in the space. Payload to match. First level is object type, the second level are elements to match. (I.e. in a person object with firstname, lastname, it would be firstname=xxx). You cannot have 2 levels.
take	...	Same parameters as read
write	timeToLiveMs json	How long the object shall live in the space TODO Shall change to duration Payload. This is the String representation of the client side JSON object.

notify	duration json	How long the notification registration shall exist. As for read
lease cancel	callId	The caller id the notification lease is registered to.

4.6. Servlet configuration parameters

If you want to exclude either take or write as feature, you can do this in the web.xml file. This is relevant when you have a service which shall provide a read-only interface to the clients, probably due to data entering the space from a backend service.

Init parameter	Significance
disableTake	If disableTake is true, clients will be unable to remove anything from the space
disableWrite	If disableWrite is true, clients will not be able to write anything to the space

The JavaScript client makes it possible to communicate with the SemiSpace from JavaScript. This JavaScript client mimics the Java interface as closely as possible and makes it possible to distribute JSON objects between the JavaScript client and the space.

Objects in the space are available as JSON objects for the JavaScript client and JSON objects pushed to the space by the JavaScript client are available for other clients. JSON objects can also be mapped directly to any known JAVA object and vise versa.

If you are not familiar with how to work against a space we recommend you read a simple introduction before you proceed.

4.7. Communication introduction

The JavaScript client rely on communicating with the space trough a CometD implementation. SemiSpace use the latest version of CometD which use a WebSocket implementation for browsers with native WebSocket support and a the long polling Bayeux protocol for browsers not supporting WebSockets.

4.8. Browser support

The client has been tested and proved to work at the following browsers:

Desktop:

- IE6 and newer
- Opera 10.x and newer
- FireFox 3.x and newer
- Chrome 5.x and newer
- Safari 3.x and newer

Mobile:
- Opera Mobile 10.x and newer
- Android 2.2 and newer
- Fennec 2.x beta and newer

4.9. Installation

Since the JavaScript client rely on communicating with the space trough a CometD implementation the JavaScript client need one of the JavaScript libraries CometD have bindings too. These libraries are Dojo Kit and jQuery .

The SemiSpace JavaScript client does only use the CometD connection module in these libraries and does not depend on any other functions in these libraries.

There are two ways to get up and running; The "custom way" and the "easy way". If your project already are depending on one of the libraries CometD have bindings to (Dojo Kit or jQuery), the "custom way" are the part you want to dive into. If you have no existing library and just want to get up and running, the "easy way" are the part you should read.

4.10. The custom way

The CometD module are not part of the core functionality in these libraries and must therfore be installed in the library of choice.

TODO: complete description....

4.11. The easy way

For your convenience the SemiSpace Comet package come with prepared builds of DojoKit and jQuery where the CometD module have been applied. To get started, select one of the provided prebuilt libraries for your project.

4.11.1. Dojo Kit

To use Dojo Kit for communication with the SemiSpace server, include Dojo Kit:

```
<script type="text/javascript" src="/semispace-comet-server/js/lib/dojo-release-1.4.3/dojo/dojo.js"> </script>
```

Then include the SemiSpace JavaScript client:

```
<script type="text/javascript" src="/semispace-comet-server/js/core/semispace.js"> </script>
```

4.11.2. jQuery

To use jQuery for communication with the SemiSpace server, include jQuery:

```
<script type="text/javascript" src="/semispace-comet-server/js/lib/jquery/jquery-1.4.2.js"> </script>
```

Then include the SemiSpace JavaScript client:

```
<script type="text/javascript" src="/semispace-comet-server/js/core/semispace.js"> </script>
```

4.12. Communicating with the space

The SemiSpace JavaScript client provides a common interface against the space so whatever JavaScript library used the methods are the same among them. The SemiSpace JavaScript client will automatically detect which JavaScript library in use so no configuration against the library are needed to get the SemiSpace JavaScript client up running.

- Connecting to the space

To make the SemiSpace JavaScript client talk to the SemiSpace server a connection need to be stabilised. This is done by initializing the connection module.

The init method can take a connection configuration object. This object are similar to the CometD configuration .

If you run on a default setup there should be no need to provide a connection configuration to the init method. The most needed parameter in the configuration to override will probably be URL the server. To override the server URL you provide a config object to the init method as follow:

When a initialization are done you can connect by executing the connection method:
- Disconnecting from the space

It's possible to disconnect from the server by executing the disconnect method:
- Listening in on the communication

It is also possible to listen in on the communication between the client (browser) and the server. This can be used to get valuable information on what is going on in the browser when communicating with the server. It can also be used to provide status messages to the user on what is going on.

You listen in on the communication by providing a callback function to the meta listener method:

The provided callback function must take two metod variables as input. The first variable will get a status code, ranging from 1 to 9, from the SemiSpace JavaScript library. The second variable will get the raw communication message provided by CometD.

A popular pattern is to listen in on the status codes and provide the user with information on the connection to the server.

The status codes are as follow

value	Description
1	Handshake is done. The client has contacted the server but no connection are set up yet
2	The client are now connected to the server. Data can now be excanged with the server
3	The client was connected to the server but has lost connection to the server
4	The client communicates with the server. This will occur pretty frequently since the client pings the server from time to time

5	The client was disconnected from the server by the disconnect method
6	The client has subscribed to a CometD metachannel
7	The client has unsubscribed from a CometD metachannel
8	The client has published a message to the server
9	The client tried to publish a message to the server but failed

4.13. Working against the space

When a connection is set up, its possible to send and recieve JSON objects to and from the space. This JavaScript client mimics the [Java interface](#) as closely as possible and are as follow:
- write
- read
- readIfExist
- take
- takeIfExist
- notify

TODO: remember release function on notify

A note on handling JSON

The JavaScript client does not stringify or parse the JSON object communicated with the space. This is left to the implementation and must be dealt with on each communication.

Each JSON object transmitted to the space must be stringified before sent.

Example:

Each JSON object received from the space must be parsed before it can be dealt with by JavaScript.

Example:

5. Usage

5.1. Space interaction

It is sometimes slightly confusing to use space based logic. This chapter contains some hints and pointers.

5.1.1. Space is not intended to be used for long term storage

A tuple space is not intended to be used for long living data. If you have needs in this direction, an ORM such as Hibernate will solve this better for you.

There is nothing wrong with mixing strategies, however.

5.1.2. Query timeout

When you are using the master / worker pattern, you will benefit from letting the query for the master answer live slightly longer the life time of the query itself.

The reason for the slightly longer life of the wait for answer, is that you need to consider the worst case time, which also includes the network traffic time, and the processing time. If you intend to have the query live shorter, decrease both timeout values, not only one.

5.1.3. Space objects are serialized

The object identity is lost when you put the objects into the space. This can be used for simplifying the queries, as you can reuse the object.

5.1.4. Make allowances for asynchronous operations

When programming against the space, try to make allowances for the operations being asynchronous. When you are treating elements, they may appear out of order as you may have more than one set of clients. If your are dependent on a certain order, you may need two keys, representing a counter and the other an operation ID.

5.1.5. Use objects and not primitives

In your holder object, do not use primitives, use objects instead. Otherwise, you will always query on the primitive value, which for int will be 0 (zero). If you use Integer for int , you can omit the value, and query on anything, as it is null.

5.1.6. Do not query on interfaces and sub class types

When you perform a query, you essentially fill out a object which shall be matched on all fields that are filled out for a given object type. The matching is not performed on class hierarchies, i.e., you cannot query with a parent class and get results of a sub class type. Interfaces are also disregarded as well.

The reason for this, is to support other languages / structures, such as ruby or php.

5.1.7. Make allowances for failure when using webservices

You need to make your application tolerate failures when running against the SemiSpace webservices, as the webservices may not always be present. The problem you try to solve, is that random outages create an exception which in effect stops your program.

Write your catch in a manner similar to the following:

```
} catch (SemiSpaceProxyException exception) {
    log.warn("Got a problem with SemiSpace connection.", exception);
    // need to sleep in order not to hammer connection, which
    // is relevant if you are in a loop.
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        // Ignored
    }
}
```

5.2. Using Spring to configure space

Spring can be used for configuration. Remember that this configuration is not interchangeable with other configuration options, due to reasons explained in the Terracotta configuration chapter.

Notice that this description is not complete, as there are several elements to configure and take into consideration. The easiest way of working with this, is probably to start with the sources for semispace-war, run the application with mvn jetty:run and examine the result(s).

5.2.1. Configure your webapp's beans for spring

As of Terracotta 3.2, you do not need to add anything particular in your tc-config.xml file.

The part you need, is the configuration in Spring's application-context.xml

```xml
<!--
The space itself. It may be distributed with terracotta, or
be stand alone.
-->
<bean id="semispace" class="org.semispace.SemiSpace" scope="singleton"
factory-method="retrieveSpace" />
```

The bean is used in the "normal" way, which is to say that it is either injected into a controller, or retrieved as a bean. Notice that you do not need to configure SemiSpace for Spring, as the other parts of the tc-config.xml covers those classes, even when they are instantiated with spring.

5.2.1.1. Exposing the web service in spring

The webservices configuration needs some more configuration in applicationContext.xml:

```xml
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

<!--
Definition of an unauthenticated space. Useful if you are
within a firewall, or otherwise do not expose the
service to the world.
-->
<jaxws:endpoint id="space" implementor="#spaceproxy"
address="/space" />

<bean id="spaceproxy"
class="org.semispace.ws.WsSpaceImpl">
```

```
<property name="space">
<ref bean="semispace" />
</property>
</bean>
```

5.2.1.2. Running the CXF servlet

In addition to setting up the spring controller servlet, you need to start the CXF servlet in web.xml:

```
<servlet>
  <servlet-name>CXFServlet</servlet-name>
  <servlet-class>
      org.apache.cxf.transport.servlet.CXFServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>CXFServlet</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

This will interface with the CXF configuration you performed in applicationContext.xml.

5.2.2. JMX exposure

The application context for semispace-war is configured to expose statistical data to a JMX client. In order to avoid setting up alternate strategies for injection of the statistical object, the data is exposed through the SemiSpace instance itself. The problem is that it is difficult (but not impossible) to successfully use different wiring strategies together with terracotta. SemiSpace uses the easiest approach...

The following spring MBean configuration is used for exposing the statistic:

```
<bean id="mbeanServer"
class="org.springframework.jmx.support.MBeanServerFactoryBean">
<!-- indicate to first look for a server -->
<property name="locateExistingServerIfPossible" value="true" />
</bean>
<!--
this bean needs to be eagerly pre-instantiated in order for the exporting to occur;
this means that it must not be marked as lazily initialized
-->
<bean id="exporter"
class="org.springframework.jmx.export.MBeanExporter">
<property name="beans">
```

```xml
<map>
<entry key="bean:name=semiSpaceStatistics"
value-ref="semispace" />
</map>
</property>
<property name="server" ref="mbeanServer" />
   <property name="assembler">
    <bean
class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
      <property name="managedMethods">
<value>numberOfSpaceElements,numberOfBlockingRead,numberOfBlockingTake,numberOf
MissedRead,numberOfMissedTake,numberOfNumberOfListeners,,numberOfRead,numberOfT
ake,numberOfWrite
   </value>
     </property>
    </bean>
   </property>
 </bean>
```

You need either a container which registers and presents the JMX data, or SDK-1.6.

5.2.2.1. Examine JMX data with JConsole

With SDK-1.6 (or greater), you have jconsole available, and can connect to the bean for obtaining statistical information.

The actor pattern framework is a powerful mechanism for inter process communication. The framework (mainly) supports communication between actors.

One of the main benefits of the abstraction is that you can easily create an asynchronous message based system, which is suitable for (particularly) a Swing application. As Swing (as almost all GUI libraries) is based on a single thread model , you need to pass any real work to a worker thread - unless you do not mind creating a sluggish application.

5.3. Utilizing actors in a Swing application

Actors in SemiSpace extend the Actor object. The object offers send , receive and the optional methods getReadTemplates and getTakeTemplates . A typical "manager" actor will register templates it want to react on. For instance, an OrderManager actor will want to receive Orders, place by a Client actor.

The following declaration is annotated with SwingActor which makes the receive call executed on the Swing thread, which is essential when you are changing values that shall be displayed in the GUI.

```
@SwingActor
public class SwingActorSearch extends Actor {
```

Sending an actor query is performed with send. The example fills out an address query template, and sends it to whichever actor that is willing to answer. The receiving actor can, of course, reside in a server process (which is the case in the tutorial).

```
AddressQuery query = new AddressQuery();
query.setAddress(address);
send( query );
```

After the "search manager" actor has performed the query, it will reply to the sender by using the

originator identification field on the payload as destination address. This makes the answer go directly to the actor which made the query, and the answer is given in in the receive method:

```
public void receive(ActorMessage msg) {
   if ( msg.isOfType(GoogleAddress.class)) {
      swingAction.setEnabled(true);
      fillArea.setText(msg.getPayload().toString());
   } else {
      log.warn("Unexpected message: "+msg.getPayload().getClass().getName());
   }
}
```

5.4. Security considerations using a Tuple Space

Access to the tuple space is necessarily read / write for all clients of the space. This is analogous to having a JDBC source which is read / write. It feels slightly different, though, as you may relate to several clients simultaneously, not only a single EAR.

It is rather easy to create malicious clients, and you therefore need to trust your client - on some some level.

When using the webservices module, you may want to enforce some additional constraints, such as allowing connection only from certain IPs. Additionally, you may want to use the authenticating token service. This will nevertheless only guarantee that the malicious user has authenticated...

The bottom line is that using the webservices module, you may want to take additional steps for securing your space. For instance, you may want to make your connections read only, and accept input to the space through some other channel (in order to avoid having rubbish inserted into your space). The other channel may be a servlet endpoint, which inserts relevant data into the space.

5.5. Issues and limitations

Disappearance of admin server
If the admin server disappears or is changed, you may experience time skew problems. This is as resynchronization takes the time from the instance identified as admin. If the difference in skew is large, you may experience some incorrect lifetimes. This is only relevant when distributing with Terracotta.

Webapp and console apps
Notice when having a heterogeneous system with a webapp and a console application, you need to configure the the applications in your tc-config.xml file. More details can be found here:
Terracotta configuration manual .

Spring loaded or class loaded
You can not mix whether you load the SemiSpace with Spring or with from the java objects directly, and need to choose one or the other.

More than one terracotta-loaded webapp on the same server
You can not deploy more than one SemiSpace-enabled war on one server, unless you use different roots in the tc-config.xml file. The error terracotta gives you is: "Perhaps you have the same root name assigned more than once to variables of different types. " This problem is due to the webapp class loader. In the tutorial project, this problem is solved by using webservices locally.

You can solve this problem with grouping your web applications in the tc-config.xml file.

Even though you can share classes between a standalone app and a webapp (and several standalone webapps), you still cannot share objects between webapps in the same container. Please see:

- https://jira.terracotta.org/jira/browse/CDV-112 (Marked as closed)
- https://jira.terracotta.org/jira/browse/CDV-272 (Marked as solved)
- https://jira.terracotta.org/jira/browse/CDV-81

A simple workaround is to use different processes for each war. The only problem with this, is that the shared wars need to be equal. This should not be necessary, but as of Terracotta version 3.0.1, I did not manage to get it to work. Even when having an app-groups configuration. This may have something to do with the spring configuration.

This boils down to the following: When having a complex setup with many different applications, make sure they can be distributed early. If you wait to long before running into a problem, you have a real risk of having to modify your archetecture on an ad hoc basis, which is suboptimal - to say the least.

You may need to download sun jars
Some of the dependencies, may demand that you have downloaded some 3rd party jars from Sun. If you only use semispace-main, this is not relevant, as it does not have any such dependencies. Read more about this in Mavens guide for coping with sun jars.

Changed fields during class upgrade will break your runtime
The data which is stored in semispace, is your objects as XML. Therefore, a change in, lets say, field names will result in an error when the object has been read from the space. If you are making changes to your object, which is not compatible with existing contents in the space, you need to remove all relevant elements from the space first.

Do not insert inner classes
When inserting inner classes, XStream will add a reference to the outer class - in toto. This can lead to unexpected and devastating results, particularly when using the persistence add-on. In the best case, you just get an unexpected increase of size. Worst case, you get severe faults due to non-serializability of the element.

Be careful with protected fields
In order to reduce visibility for a (transport) class, it may be tempting to change a field to protected. This will make SemiSpace disregard the field in total, which leads to confusing query results, as the field would not be part of a query. Remember: All fields that are going to be used for queries need to be public. However, you can have as many private and protected fields as you like - as long as they are not part of a query.

XML will expose all variables
When transporting an object over the web services interface, all fields will be exposed, as XML does not differentiate between private and public fields. Bear this in mind when constructing transport objects, as you could introduce side effects when querying on the space with XML.