Fault Injection framework: HOW TO use it, test using artificial faults, and develop new faults.

Konstantin Boudnik

- Fault Injection framework: HOW TO use it, test using artificial faults, and develop new faults.
  - Introduction
  - Assumptions
  - Architecture of the fault injection framework
    - Configuration management
    - Probability model
    - Fault injection mechanism: AOP and AspectJ
    - Existing join points
  - Aspects examples
  - Fault naming convention & namespaces
  - Development tools
  - Putting it all together
    - How to use fault injection framework
  - Additional information and contacts

Introduction

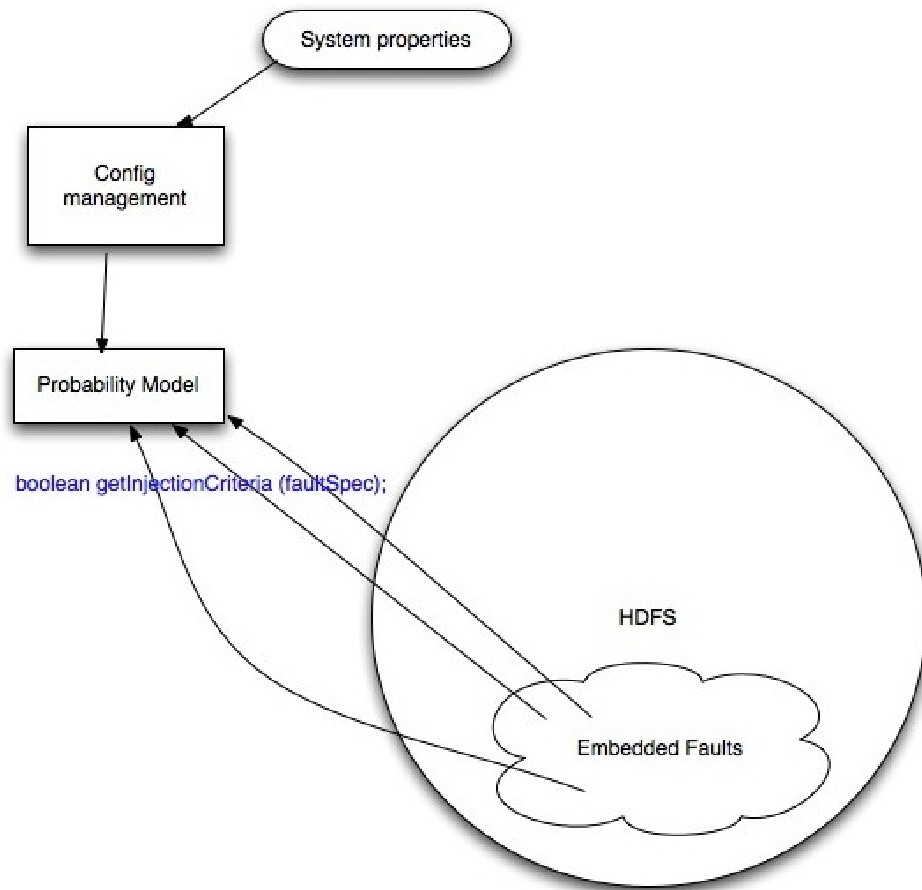
The following is brief help for Hadoops' Fault Injection framework and a developer's guide for those who will be developing their own faults (aspects).

An idea of fault injection (FI) is fairly simple: it is an infusion of errors and exceptions into an application's logic to achieve a higher coverage and fault tolerance of the system. Different implementations of this idea are available at this day. Hadoop's FI framework is built on top of Aspect Oriented Paradigm (AOP) implemented by AspectJ toolkit.

Assumptions

Current implementation of the framework assumes that faults it will be emulating are of non-deterministic nature. I.e. the moment of a fault's happening isn't none in advance and is a coin flip based.

Architecture of the fault injection framework



Configuration management

This piece of the framework allow to set expectations for faults to happen. The settings could be applied either statically (in advance) or in a runtime. There's two ways to configure desired level of faults in the framework:

- editing `src/aop/fi-site.xml` configuration file. This file is similar to other Hadoop's config files
- setting system properties of JVM through VM startup parameters or in `build.properties` file

Probability model

This fundamentally is a coin flipper. The methods of this class are getting a random number between `0.0` and `1.0` and then checking if new number has happened to be in the range of `0.0` and a configured level for the fault in question. If that condition is true then the fault will occur.

Thus, to guarantee a happening of a fault one needs to set an appropriate level to `1.0`. To completely prevent a fault from happening its probability level has to be set to `0.0`

Nota bene: default probability level is set to `0` (zero) unless the level is changed explicitly through the configuration file or in the runtime. The name of the default level's configuration parameter is `fi.*`

Fault injection mechanism: AOP and AspectJ

In the foundation of Hadoop's fault injection framework lays cross-cutting concept implemented by AspectJ. The following basic terms are important to remember:

- **A cross-cutting concept** (aspect) is behavior, and often data, that is used across the scope of a piece of software
- In AOP, the **aspects** provide a mechanism by which a cross-cutting concern can be specified in a modular way
- **Advice** is the code that is executed when an aspect is invoked
- **Join point** (or pointcut) is a specific point within the application that may or not invoke some advice

Existing join points

The following readily available join points are provided by AspectJ:

- Join when a method is called
- Join during a method's execution
- Join when a constructor is invoked
- Join during a constructor's execution
- Join during aspect advice execution
- Join before an object is initialized
- Join during object initialization
- Join during static initializer execution
- Join when a class's field is referenced
- Join when a class's field is assigned
- Join when a handler is executed

Aspects examples

```
package org.apache.hadoop.hdfs.server.datanode;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.fi.ProbabilityModel;
import org.apache.hadoop.hdfs.server.datanode.DataNode;
import org.apache.hadoop.util.DiskChecker.*;

import java.io.IOException;
import java.io.OutputStream;
import java.io.DataOutputStream;

/**
* This aspect takes care about faults injected into datanode.BlockReceiver
* class
*/
public aspect BlockReceiverAspects {
	public  static  final  Log LOG = LogFactory.getLog(BlockReceiverAspects.class);

	public  static  final  String  BLOCK_RECEIVER_FAULT="hdfs.datanode.BlockReceiver";
	pointcut callReceivePacket() :
	call (* OutputStream.write(..))
	&& withincode (* BlockReceiver.receivePacket(..))
	// to further limit an application of this aspect a very narrow 'target' can be used as follows
	//  && target(DataOutputStream)
	&& !within(BlockReceiverAspects +);

	before () throws  IOException  : callReceivePacket () {
		if  (ProbabilityModel.injectCriteria(BLOCK_RECEIVER_FAULT)) {
			LOG.info("Before the injection point");
			Thread.dumpStack();
			throw  new  DiskOutOfSpaceException ("FI: injected fault point at " +
			thisJoinPoint.getStaticPart( ).getSourceLocation());
		}
	}
}
```

The aspect has two main parts: the join point `pointcut callReceivePacket()` which servers as an identification mark of a specific point (in control and/or data flow) in the life of an application. A call to the advice - `before () throws IOException : callReceivePacket()` - will be injected before that specific spot of the application's code.

The pointcut identifies an invocation of class' `java.io.OutputStream write()` method with any number of parameters and any return type. This invoke should take place within the body of method `receivePacket()` from class BlockReceiver. The method can have any parameters and any return type. Possible invocations of `write()` method happening anywhere within the aspect `BlockReceiverAspects` or its heirs will be ignored.

Note 1: This short example doesn't illustrate the fact that you can have more than a single injection point per class. In such a case the names of the faults have to be different if a developer wants to trigger them separately.

Note 2: After injection step you can verify that the faults were properly injected by searching for `ajc` keywords in a disassembled class file.

Fault naming convention & namespaces

For the sake of unified naming convention the following two types of names are recommended for a new aspects development:

- Activity specific notation (as when we don't care about a particular location of a fault's happening). In this case the name of the fault is rather abstract: `fi.hdfs.DiskError`
- Location specific notation. Here, the fault's name is mnemonic as in: `fi.hdfs.datanode.BlockReceiver[optional location details]`

Development tools

- Eclipse [AspectJ Development Toolkit](#) might help you in the aspects' development process.
- IntelliJ IDEA provides AspectJ weaver and Spring-AOP plugins

Putting it all together

Faults (or aspects) have to injected (or woven) together before they can be used. Here's a step-by-step instruction how this can be done.

Weaving aspects in place:
```
% ant injectfaults
```
If you misidentified the join point of your aspect then you'll see a warning similar to this one below when 'injectfaults' target is completed:
```
[iajc] warning at src/test/aop/org/apache/hadoop/hdfs/server/datanode/BlockReceiverAspects.aj:44::0
advice defined in org.apache.hadoop.hdfs.server.datanode.BlockReceiverAspects has not been applied
[Xlint:adviceDidNotMatch]
```
It isn't an error, so the build will still be successful.

To prepare dev.jar file with all your faults weaved in place run (HDFS-475 pending)
```
% ant jar-fault-inject
```

Test jars can be created by
```
% ant jar-test-fault-inject
```

To run HDFS tests with faults injected:
```
% ant run-test-hdfs-fault-inject
```

How to use fault injection framework

Faults could be triggered by the following two meanings:

- In the runtime as:

```
% ant run-test-hdfs -Dfi.hdfs.datanode.BlockReceiver=0.12
```

To set a certain level, e.g. 25%, of all injected faults one can run

```
% ant run-test-hdfs-fault-inject -Dfi.*=0.25
```

- or from a program as follows:

```
package org.apache.hadoop.fs;

import org.junit.Test;
import org.junit.Before;
import junit.framework.TestCase;

public class DemoFiTest extends TestCase {
        public static final String BLOCK_RECEIVER_FAULT="hdfs.datanode.BlockReceiver";
        @Override
        @Before
        public void setUp(){
                //Setting up the test's environment as required
        }

        @Test
        public void testFI() {
                // It triggers the fault, assuming that there's one called
'hdfs.datanode.BlockReceiver'
                System.setProperty("fi." + BLOCK_RECEIVER_FAULT, "0.12");
                //
                // The main logic of your tests goes here
                //
                // Now set the level back to 0 (zero) to prevent this fault from happening again
                System.setProperty("fi." + BLOCK_RECEIVER_FAULT, "0.0");
                // or delete its trigger completely
                System.getProperties().remove("fi." + BLOCK_RECEIVER_FAULT);
        }

        @Override
        @Test
        public void tearDown() {
                //Cleaning up test test environment
        }
}
```

as you can see above these two methods do the same thing. They are setting the probability level of hdfs.datanode.BlockReceiver at 12%. The difference, however, is that the program provides more flexibility and allows to turn a fault off when a test doesn't need it anymore.

Additional information and contacts

This two sources of information seem to be particularly interesting and worth further reading:

- http://www.eclipse.org/aspectj/doc/next/devguide/
- AspectJ Cookbook (available through Safari website (subscription is required), or in hard copy form ISBN-13: 978-0-596-00654-9)

Should you have any farther comments or questions to the author check HDFS-435