Routing and Failing Strategies

Routing and Failing Strategies

This page provides examples for different routing and failing setups that are possible with DistributeMe.

Failover

Failover strategies include switching from one instance to another in case the first instance isn't answering anymore. There are two primary straight failover strategies and a lot of mixups with other strategies.

Failover on error but return.

This strategy includes failing over to a backup instance but try to connect to original instance on each call.

To enable this strategy you have to do following:

1. Mark your method or the whole class as failover:

```
	@FailBy(strategyClass=Failover.class)
	void aMethod(...);
```

This will enable your code to failover to another instance. The only thing you require is to start both instances.

Primary instance:

```
./start.sh <package>.generated.XYZServer
```

Backup instance:

```
./start.sh -DdimeRegistrationNameProvider=org.distributeme.core.failing.Failover <package>.generated.XYZServer
```

This additional Parameter will force the server process to add '*.failover*' to the registration string, which is also added for each retry-call by the strategy.

You can see the result in the registry:

```
<services>
<service serviceId="org_distributeme_test_fail_FailableService" host="192.168.200.107" port="9250" protocol="
rmi" instanceId="nfhadelfbm" globalId="rmi://org_distributeme_test_fail_FailableService" registrationString="
rmi://org_distributeme_test_fail_FailableService.nfhadelfbm@192.168.200.107:9250"/>
<service serviceId="org_distributeme_test_fail_FailableService-failover" host="192.168.200.107" port="9250"
protocol="rmi" instanceId="mfmmnntudo" globalId="rmi://org_distributeme_test_fail_FailableService-failover"
registrationString="rmi://org_distributeme_test_fail_FailableService-failover.mfmmnntudo@192.168.200.107:9250"/>
</services>
```

Note: this method will produce additional traffic on the registry, because each 'first' call will try to lookup the original instance first. On the plus side, as soon as the instance is restarted the calls will go directly to this instance.

Failover on error and stay.

This strategy forces the client to switch to backup instance permanently.

1. Mark your method or the whole class as failover:

```
	@Route(routerClass=Failover.class)
	@FailBy(strategyClass=Failover.class)
	void aMethod(...);
```

Start the servers as in the above example and you are done. The difference between both approaches is that the additional router directive will route all calls to the failover-instance once one call to the original instance failed. This is faster, because there is no calls to registry anymore, but you will never return to the original instance without further measures.

Here is the code of the Failover class:

SVN: **/distributeme/trunk/distributeme-core/java/org/distributeme/core/failing/Failover.java**

Unknown macro: {groovy}

```
def data = new URL('http://svn.anotheria.net/opensource//distributeme/trunk/distributeme-core/java/org/distributeme/core/failing/Failover.java').getText()
println '
```

```
' + data + '
```

```
'
```

Retry

There are a lot of different scenarios for retry. Strictly speaking is failover also a form of retry. But there are other:

Retry forever

The simpliest of Retry strategies is to simply retry the call. This is just a plain declaration in the service interface.

```
@FailBy(strategyClass=RetryCall.class)
void aMethod(...);
```

Of course this approach has limited use. In a high traffic application you will pretty soon run out of threads, cause the retry blocks the executing thread for even longer. Another limit is given through the implementation of distributeme failing - it is based on re-calling the same method again and again. After some period of time your stack size will be depleted and a StackOverflowException will rush back the line.

Make a pause and retry

Due the limits of the previous example, it's recommendable to add some sleep time to the call, allowing the machine to perform other tasks.

```
@FailBy(strategyClass=WaitOneSecondAndRetry.class)
void aMethod(...);
```

As derivable from the name, this failing strategy forces the active thread to sleep a second before retrying the call, hence giving the server a real chance to recover or become restarted. Still the problems of the previous example remain. A Stackoverflow exception will come much slower, but it will, probably after 16 minutes.

Of course you are free to implement strategies that sleep another amount of time, or wait for something, or... Here's the code of WaitOneSecondAndRetry. class:

SVN: **/distributeme/trunk/distributeme-core/java/org/distributeme/core/failing/WaitOneSecondAndRetry.java**

Unknown macro: {groovy}

```
def data = new URL('http://svn.anotheria.net/opensource//distributeme/trunk/distributeme-core/java/org/distributeme/core/failing/WaitOneSecondAndRetry.
java').getText()
println '
```

```
' + data + '
```

```
'
```

RetryOnce

The above strategies are also imaginable in combination with RetryOnce