Hadoop:
Code Injection
Distributed Fault Injection

Konstantin Boudnik
Hadoop committer, Pig contributor
cos@apache.org

Few assumptions

- The following work has been done with use of AOP injection technology called AspectJ
- Similar results could be achieved with
  - direct code implementation
  - MOP (monkey-patching)
  - Direct byte-code manipulations
- Offered approaches aren't limited by the scope of Hadoop platform ;)
- The scope of the talk isn't about AspectJ nor AOP/MOP technology

Code Injection

What for?

- Some APIs as extremely useful as dangerous if made public
  - stop/blacklist a node or daemon
  - change a node configuration
- certain functionality is experimental and needn't to be in production
- a component's source code is unavailable
- a build's re-spin isn't practical
- many changes of the same nature need to be applied
- your application doesn't have enough bugs yet

Use cases

- producing a build for developer's testing
- simulate faults and test error recovery before deployment
- to sneak-in to the production something your boss don't need to know

Injecting away

```
pointcut execGetBlockFile() :
// the following will inject faults inside of the method in question
  execution (* FSDataset.getBlockFile(..)) && !within(FSDatasetAspects +);

// This aspect specifies the logic of our fault point.
// In this case it simply throws DiskErrorException before invoking
// the method, specified by callGetBlockFile() pointcut
before() throws DiskErrorException : execGetBlockFile() {
  if (ProbabilityModel.injectCriteria(FSDataset.class.getSimpleName())) {
    LOG.info("Before the injection point");
    Thread.dumpStack();
    throw new DiskErrorException("FI: injected fault point at "
        + thisJoinPoint.getStaticPart().getSourceLocation());
  }
}
```

Injecting away (intercept & mock)

```
pointcut callCreateUri() : call (URI FileDataServlet.createUri(
    String, HdfsFileStatus, UserGroupInformation, ClientProtocol,
    HttpServletRequest, String));

/** Replace host name with "localhost" for unit test environment. */
URI around () throws URISyntaxException : callCreateUri() {
  final URI original = proceed();
  LOG.info("FI: original uri = " + original);
  final URI replaced = new URI(original.getScheme(),
      original.getUserInfo(),
      "localhost", original.getPort(), original.getPath(),
      original.getQuery(),
      original.getFragment()) ;
  LOG.info("FI: replaced uri = " + replaced);
  return replaced;
}
```

Distributed Fault Injection

Why Fault Injection

- Hadoop deals with many kinds of faults
  - Block corruption
  - Failures of disk, Datanode, Namenode, Clients, Jobtracker, Tasktrackers and Tasks
  - Varying rates of bandwidth and latency
- These are hard to test
  - Unit tests mostly deal with specific single faults or patterns
  - Faults do not occur frequently and hard to reproduce
- Need to inject fault in the real system (as opposed to a simulated system)
- More info
  - http://wiki.apache.org/hadoop/HowToUseInjectionFramework

Usage models

- An actor configures a Hadoop cluster and “dials-in” a desired faults then runs a set of applications on the cluster.
  - Test the behavior of particular feature under faults
  - Test time and consistency of recovery at high rate of faults
  - Observe loss of data under certain pattern and frequency of faults
  - Observe performance/utilization
    - Note: can inject faults in the real system's (as opposed to a simulated system) running jobs
- An actor write/reuse a unit/function test using the fault inject framework to introduce faults during the test
- Recovery procedures testing (!)

Fault examples (Hdfs)

- Link/communication failure and communication corruption
  - Namenode to Datanode communication
  - Client to Datanode communications
  - Client to Namenode communications
- Namenode related failures
  - General slow downs
  - Edit logs slow downs
  - NFS-mounted volume is slow or not responding
- Datanode related failures
  - Hardware corruption and data failures
- Storage latencies and bandwidth anomalies

Fault examples (Mapreduce)

- Task tracker
  - Lost task trackers
- Tasks
  - Timeouts
  - Slow downs
  - Shuffle failures
  - Sort/merge failures
- Local storage issues
- JobTracker failures
- Link communication failures and corruptions

Scale^{n}

- Multi-hundred nodes cluster
- Heterogeneous environment
  - OS. switches, secure/non-secure configurations
- Multi-node faults scenarios (e.g. pipelines recovery)
- Requires fault manager/dispensary
  - Support for multi-node, multi-conditions faults
  - Fault identification, reproducibility, repeatability
  - Infrastructure auto-discovery to avoid configuration complexities

Coming soon...

Client side

```
pointcut execGetBlockFile() :
// the following will inject faults inside of the method in question
  execution (* FSDataset.getBlockFile(..)) && !within(FSDatasetAspects +);

before() throws DiskErrorException : execGetBlockFile() {
  ArrayList<GenericFault> pipelineFault =
       FiDispenser.getFaultsFor(FSDataset.class,
       FaultID.PipelineRecovery(),
       RANDOM);

   for (int i = 0; i < pipelineFault.size(); i++) {
     pipelineFault.get(i).execute();
   }
}
```

Fault dispenser

```
MachineGroup Rack1DataNodes = new MachineGroup(rack1, TYPE.DN)

Rack1DataNodes.each {
  if (it.type == RANDOM) {
    it.setTimeout(random.nextInt(2000))
    it.setType(DiskErrorException.class)
    it.setReport('logcollector.domain.com', SYSLOG)
  }
}
```

Q & A

Attic slides

White-box system testing: Herriot

Goals

- Write cluster-based tests using Java object model
- Automate many types of tests on real clusters:
  - Functional
  - System
  - Load
  - Recovery
- More information
  - http://wiki.apache.org/hadoop/HowToUseSystemTestFramework

Main Features

- Remote daemon Observability and Controllability APIs
- Enables large cluster-based tests written in Java using JUnit (TestNG) framework
- Herriot is comprised of a library of utility APIs, and code injections into Hadoop binaries
- Assumes a deployed and instrumented cluster
- Production build contains NO Herriot instrumentation
- Supports fault injection

Major design considerations

- Common
- RPC-based utilities to control remote daemons
- Daemons belong to different roles
- Remote process management from Java: start/stop, change/push configuration, etc.
- HDFS and MR specific APIs on top of Common

Common Features

- Get a daemon (a remote Hadoop process) current configuration
- Get a daemon process info: thread#, heap, environment…
- Ping a daemon (make sure it's up and ready)
- Get/list FileStatus of a path from a remote daemon
- Deamon Log Inspection: Grep remote logs, count exceptions…
- Cluster setup/tear down; restart
- Change a daemon(s) configuration, push new configs…

Deployment Diagram