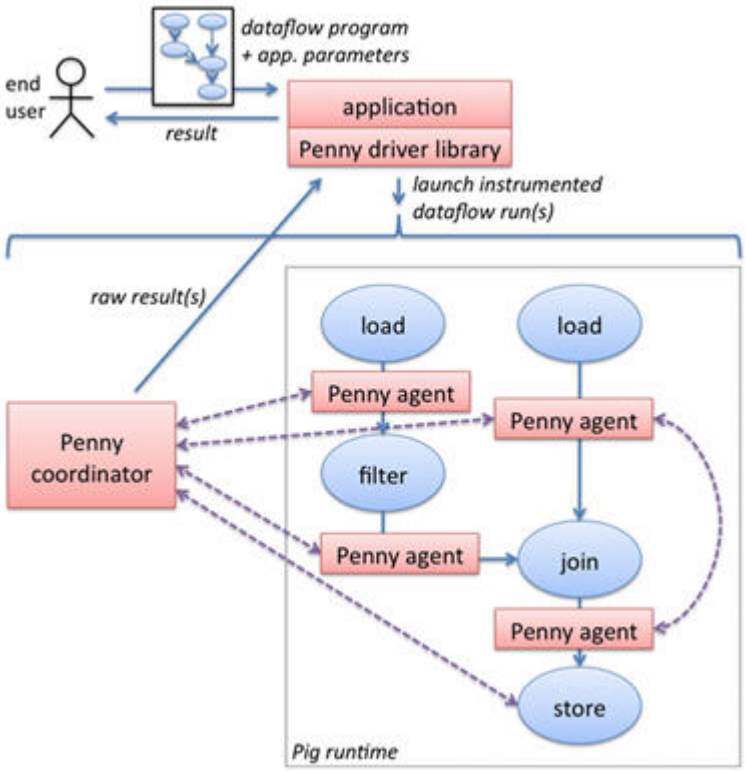
PennyCreateYourOwnTool

Creating your own Penny tool

Penny makes it easy to create custom monitoring and debugging tools for Pig. Here's how:

How Penny works

Before you can write a tool, you need a bit of background on how Penny instruments Pig scripts (called "dataflow programs" in the following diagram).



As shown in this diagram, Penny inserts one or more "monitor agents" between steps of the pig script, which observe data flowing between the pig script steps. Monitor agents run arbitrary Java code as needed for your tool, which has access to some primitives for tagging records and communicating with other agents and with a central "coordinator" process. The coordinator also runs arbitrary code defined by your tool.

The whole thing is kicked off by the tool's Main program (labeled "application" in the diagram), which receives instructions from the user (e.g. "please figure out why this pig script keeps crashing"), launches one or more runs of the pig script instrumented with Penny monitor agents, and reports the outcome back to the user (e.g. "the crash appears to be caused by one of these records: ...").

API

You need to write three Java classes: a Main class, a Coordinator class, and a MonitorAgent class (for certain, fancy tools, you may need multiple MonitorAgent classes). You can find many examples of Main/Coordinator/MonitorAgent classes that define Penny tools in the Penny source code (svn://research6.corp.yahoo.com/Penny/src) under org.apache.pig.penny.apps. All of the tools described in PennyToolLibrary are written using this API, so you've got plenty of examples to work with. We'll paste a few code fragments below to get you going – in fact the entire code for the "data samples" tool (all 97 lines of Java) is pasted in this twiki.

Main class

Your Main class is the "shell" of your application. It receives instructions from the user, and configures and launches one or more Penny-instrumented runs of the user's pig script.

You talk to Penny via the PennyServer class. You can do two things: (1) parse a user's pig script, (2) launch an Penny-instrumented run of the pig script. Here is the Main class for the data samples tool, described at PennyToolLibrary:

```
import java.util.HashMap;
import java.util.Map;
import org.apache.pig.penny.ClassWithArgs;
import org.apache.pig.penny.ParsedPigScript;
import org.apache.pig.penny.PennyServer;

/**
 * Data samples app.
 */
public class Main {
    public static void main(String[] args) throws Exception {
        PennyServer pennyServer = new PennyServer();
        String pigScriptFilename = args[0];
        ParsedPigScript parsedPigScript = pennyServer.parse(pigScriptFilename);
        Map<String, ClassWithArgs> monitorClasses = new HashMap<String, ClassWithArgs>();
        for (String alias : parsedPigScript.aliases()) {
            monitorClasses.put(alias, new ClassWithArgs(DSMonitorAgent.class));
        }
        parsedPigScript.trace(DSCoordinator.class, monitorClasses);
    }
}
```

The "monitorClasses" map dictates which monitor agent (if any) to place after each dataflow step (steps are identified by pig script aliases). You can also pass arguments to each monitor agent, and/or to the coordinator, as shown in this example (for the data histograms tool):

```
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import org.apache.pig.penny.ClassWithArgs;
import org.apache.pig.penny.ParsedPigScript;
import org.apache.pig.penny.PennyServer;

/**
 * Data summaries app. that computes a histogram of one of the fields of one of the intermediate data sets.
 */

public class Main {
    public static void main(String[] args) throws Exception {
        PennyServer pennyServer = new PennyServer();
        String pigScriptFilename = args[0];
        ParsedPigScript parsedPigScript = pennyServer.parse(pigScriptFilename);
        String alias = args[1]; // which alias to create histogram for
        int fieldNo = Integer.parseInt(args[2]); // which field to create histogram for
        int min = Integer.parseInt(args[3]); // min field value
        int max = Integer.parseInt(args[4]); // max field value
        int bucketSize = Integer.parseInt(args[5]); // histogram bucket size
        if (!parsedPigScript.aliases().contains(alias)) throw new IllegalArgumentException("No such alias.");
        Map<String, ClassWithArgs> monitorClasses = new HashMap<String, ClassWithArgs>();
        monitorClasses.put(alias, new ClassWithArgs(DHMonitorAgent.class, fieldNo, min, max, bucketSize));
        TreeMap<Integer, Integer> histogram = (TreeMap<Integer, Integer>) parsedPigScript.trace(DHCoordinator.
class, monitorClasses);
        System.out.println("Histogram: " + histogram);
    }
}
```

MonitorAgent class

Monitor agents implement the following API:

```
 /**
  * Furnish set of fields to monitor. (Null means monitor all fields ('*').)
  * /
 public abstract Set<Integer> furnishFieldsToMonitor(); /**
  * Initialize, using any arguments passed from higher layer.
  * /
 public abstract void init(Serializable[] args);
 /**
  * Process a tuple that passes through the monitoring point.
  *
  * @param t   the tuple
  * @param tag t's tags
  * @return FILTER_OUT to remove the tuple from the data stream; NO_TAGS to let it pass through and not give it
any tags; a set of tags to let it pass through and assign those tags
  */
 public abstract Set<String> observeTuple(Tuple t, Set<String> tags) throws ExecException;
 /**
  * Process an incoming (synchronous or asynchronous) message.
  */
 public abstract void receiveMessage(Location source, Tuple message);
 /**
  * No more tuples are going to pass through the monitoring point. Finish any ongoing processing.
  */
 public abstract void finish();
```

Here's an example from the "data samples" tool:

```
import java.io.Serializable; import java.util.Set;

import org.apache.pig.backend.executionengine.ExecException; import org.apache.pig.data.Tuple; import org.
apache.pig.penny.Location; import org.apache.pig.penny.MonitorAgent;

public class DSMonitorAgent extends MonitorAgent {

    private final static int NUM_SAMPLES = 5;
    private int tupleCount = 0;
    public void finish() { }
    public Set<Integer> furnishFieldsToMonitor() {
        return null;
    }
    public void init(Serializable[] args) { }
    public Set<String> observeTuple(Tuple t, Set<String> tags) throws ExecException {
        if (tupleCount++ < NUM_SAMPLES) {
            communicator().sendToCoordinator(t);
        }
        return tags;
    }
    public void receiveMessage(Location source, Tuple message) { }
}
```

Monitor agents have access to a "communicator" object, which is the gateway for sending messages to other agents or to the coordinator. The communicator API is:

```
 /**
  * Find out my (physical) location.
  * /
 public abstract Location myLocation();
 /**
  * Send an message to the coordinator, asynchronously.
  * /
 public abstract void sendToCoordinator(Tuple message);
 /**
  * Send a message to immediate downstream neighbor(s), synchronously.
  * If downstream neighbor(s) span a task boundary, all instances will receive it; otherwise only same-task
instances will receive it.
  * If there is no downstream neighbor, an exception will be thrown.
  * /
 public abstract void sendDownstream(Tuple message) throws NoSuchLocationException;
 /**
  * Send a message to immediate upstream neighbor(s), synchronously.
  * If upstream neighbor(s) are non-existent or span a task boundary, an exception will be thrown.
  * /
 public abstract void sendUpstream(Tuple message) throws NoSuchLocationException;
 /**
  * Send a message to current/future instances of a given logical location.
  * Instances that have already terminated will not receive the message (obviously).
  * Instances that are currently executing will receive it asynchronously (or perhaps not at all, if they
terminate before the message arrives).
  * Instances that have not yet started will receive the message prior to beginning processing of tuples.
  * /
 public abstract void sendToAgents(LogicalLocation destination, Tuple message) throws NoSuchLocationException;
// The following methods mirror the ones above, but take care of packaging a list of objects into a tuple
(you're welcome!) ...
 public void sendToCoordinator(Object ... message) {
  . sendToCoordinator(makeTuple(message));
 }
 public void sendDownstream(Object ... message) throws NoSuchLocationException {
  . sendDownstream(makeTuple(message));
 }
 public void sendUpstream(Object ... message) throws NoSuchLocationException {
  . sendUpstream(makeTuple(message));
 }
 public void sendToAgents(LogicalLocation destination, Object ... message) throws NoSuchLocationException {
  . sendToAgents(destination, makeTuple(message));
 }
```

That's pretty much it for monitor agents. Pretty simple, eh?

Coordinator class

Your tool's coordinator implements the following API:

```
 /**
  * Initialize, using any arguments passed from higher layer.
  * /
 public abstract void init(Serializable[] args);
 /**
  * Process an incoming (synchronous or asynchronous) message.
  * /
 public abstract void receiveMessage(Location source, Tuple message); /**
  * The data flow has completed and all messages have been delivered. Finish processing.
  *   * @return              final output to pass back to application
  * /
 public abstract Object finish();
```

The coordinator for the "data samples" tool is:

```
import java.io.Serializable;
import org.apache.pig.data.Tuple;
import org.apache.pig.penny.Coordinator;
import org.apache.pig.penny.Location;
public class DSCoordinator extends Coordinator {
    public void init(Serializable[] args) { }
    public Object finish() {
        return null;
    }
    public void receiveMessage(Location source, Tuple message) {
        System.out.println("*** SAMPLE RECORD AT ALIAS " + source.logId() + ": " + truncate(message));
    }
    private String truncate(Tuple t) {
        String s = t.toString();
        return s.substring(0, Math.min(s.length(), 100));
    }
}
```

Again, simple!