Swift Cookbook

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.92	March 2011		ST

Contents

1 Overview **1**

2 Swift Basics **1**

2.1 Installation 1

2.1.1 Prerequisites 1

2.2 Setting up to run Swift 1

2.2.1 Environment Setup 1

2.2.2 Setting transformation catalog 2

2.2.3 Setting Swift configuration 2

2.2.4 Setting sites.xml 3

2.3 First Swift script 3

2.4 second Swift script 3

2.5 Swift Commandline Options 4

2.6 What if Swift hangs 4

2.7 Resuming a stopped or crashed Swift Run 4

2.8 Passing an array to swift? 5

2.8.1 Mappers 5

3 Swift on Computational Infrastructures **7**

3.1 Introduction 7

3.2 Beagle 8

3.3 PADS 8

3.4 Fusion 9

3.5 OSG 9

3.6 Bionimbus 10

4 Debugging Swift **11**

5 Log Processing **12**

5.1 Make a basic load plot from Coasters Cpu log lines 12

5.1.1 Make a basic job completion plot from Coasters Cpu log lines 12

5.1.2 Make a basic Block allocation plot from Coasters Block log lines 13

5.2 Problem Reporting 13

1 Overview

This cookbook covers recipes involving setting up and running Swift under various computational systems based on the application requirements and the underlying infrastructures. The Swift framework comprises of a scripting language and the Swift runtime system. For introductory material, consult the Swift tutorial found here.

Warning
Note that the cookbook is under construction and the recipes presented here are not tested rigorously.

2 Swift Basics

2.1 Installation

This section takes you through the installation of the Swift system on your computer. We will start with the prerequisites as explained in the subsequent section.

2.1.1 Prerequisites

Check your Java Swift is a Java application. Make sure you are running Java version 5 or higher. You can make sure you have Java in your $PATH (or file such as $HOME/.soft depending upon your environment)

Following are some possible ways to detect and run Java:

```
$ grep java $HOME/.soft
#+java-sun # Gives you Java 5
+java-1.6.0_03-sun-r1
$ which java
/soft/java-1.6.0_11-sun-r1/bin/java
$ java -version
java version "1.6.0_11"
Java(TM) SE Runtime Environment (build 1.6.0_11-b03)
Java HotSpot(TM) 64-Bit Server VM (build 11.0-b16, mixed mode)
```

2.2 Setting up to run Swift

This is simple. We will be using a pre-compiled version of Swift that can be downloaded from here. Download and untar the latest precompiled version as follows:

```
$ tar zxf swift-0.92.1.tar.gz
```

2.2.1 Environment Setup

The examples were tested with Java version 1.6. Make sure you do not already have Swift in your PATH. If you do, remove it, or remove any +swift or @swift lines from your $HOME/.soft or $HOME/.bash_profile file. Then do:

```
PATH=$PATH:/path/to/swift/bin
```

Note that the environment will be different when using Swift from prebuilt distribution (as above) and trunk. The PATH setup when using swift from trunk would be as follows:

```
PATH=$PATH:/path/to/swift/dist/swift-svn/bin
```

Warning
Do NOT set SWIFT_HOME or CLASSPATH in your environment unless you fully understand how these will affect Swift's execution.

To execute your Swift script on a login host (or "localhost") use the following command:

```
swift -tc.file tc somescript.swift
```

2.2.2 Setting transformation catalog

The transformation catalog lists where application executables are located on remote sites.

By default, the site catalog is stored in etc/tc.data. This path can be overridden with the tc.file configuration property, either in the Swift configuration file or on the command line.

The format is one line per executable per site, with fields separated by tabs or spaces.

Some example entries:

```
localhost  echo    /bin/echo       INSTALLED       INTEL32::LINUX  null
TGUC       touch   /usr/bin/touch  INSTALLED       INTEL32::LINUX GLOBUS::maxwalltime ↩
    ="00:00:10"
```

The fields are: *site*, *transformation-name*, *executable-path*, *installation-status*, *platform*, and *profile* entries.

The *site* field should correspond to a site name listed in the sites catalog.

The *transformation-name* should correspond to the transformation name used in a Swift script app procedure.

The *executable-path* should specify where the particular executable is located on that site.

The *installation-status* and *platform* fields are not used. Set them to **INSTALLED** and **INTEL32::LINUX** respectively.

The *profiles* field should be set to null if no profile entries are to be specified, or should contain the profile entries separated by semicolons.

2.2.3 Setting Swift configuration

Many configuration properties could be set using the Swift configuration file. We will not cover them all in this section. see here for details. In this section we will cover a simple configuration file with the most basic properties.

```
# A comment
wrapperlog.always.transfer=true
sitedir.keep=true
execution.retries=1
lazy.errors=true
status.mode=provider
use.provider.staging=true
provider.staging.pin.swiftfiles=false
clustering.enabled=false
clustering.queue.delay=10
clustering.min.time=86400
foreach.max.threads=100
provenance.log=true
```

2.2.4 Setting sites.xml

sites.xml specifies details of the sites that Swift can run on. Following is an example of a simple sites.xml file entry for running Swift on local environment:

```
<pool handle="localhost">
<filesystem provider="local" />
<execution provider="local" />
<workdirectory >/var/tmp</workdirectory>
<profile namespace="karajan" key="jobThrottle">.07</profile>
<profile namespace="karajan"
key="initialScore">100000</profile>
</pool>
```

2.3 First Swift script

Your first Swift script Hello Swift-World!

A good sanity check that Swift is set up and running OK locally is this:

```
$ which swift

/home/wilde/swift/src/stable/cog/modules/swift/dist/swift-svn/bin/swift

$ echo 'trace("Hello, Swift world!");' >hello.swift

$ swift hello.swift

Swift svn swift-r3202 cog-r2682

RunID: 20100115-1240-6xhzxuz3

Progress:

Swift trace: Hello, Swift world!

Final status:

$
```

A good first tutorial in using Swift is at: http://www.ci.uchicago.edu/swift/guides/tutorial.php. Follow the steps in that tutorial to learn how to run a few simple scripts on the login host.

2.4 second Swift script

Following is a more involved Swift script.

```
type file;

app (file o) cat (file i)
{
    cat @i stdout=@o;
}

file out[]<simple_mapper; location="outdir", prefix="f.",suffix=".out">;

foreach j in [1:@toint(@arg("n","1"))] {

    file data<"data.txt">;
    out[j] = cat(data);
}
```

2.5 Swift Commandline Options

A description of Swift Commandline Options

Also includes a description of Swift inputs and outputs.

2.6 What if Swift hangs

Owing to its multi-threaded architecture it is often the case that the underlying java virtual machine gets into deadlock situations or Swift hangs because of other complications in its operations. Under such situations, Swift *hang-checker* chips in and gives indication of the situation.

1. how to use the information to identify and correct the deadlock.
2. How close to the Swift source code can we make the hang-checker messages, so that the user can relate it to Swift functions, expressions, and ideally source code lines?
3. The Hang Checker output is nice and useful:

```
Registered futures:
Rupture[] rups  Closed, 1 elements, 0 listeners
Variation vars - Closed, no listeners
SgtDim sub - Open, 1 listeners
string site  Closed, no listeners
Variation[] vars  Closed, 72 elements, 0 listeners
```

2.7 Resuming a stopped or crashed Swift Run

I had a .rlog file from a Swift run that ran out of time. I resumed the run using the -resume flag described in section 16.2 of the Swift User Guide and it picked up where it left off. Then I killed it because I wanted to make changes to my sites file.

```
. . .
Progress:  Selecting site:1150  Stage in:55  Active:3  Checking status:1
Stage out:37  Finished in previous run:2462  Finished successfully:96
Progress:  Selecting site:1150  Stage in:55  Active:2  Checking status:1
Stage out:38  Finished in previous run:2462  Finished successfully:96
Cleaning up...
Shutting down service at https://192.5.86.6:54813
Got channel MetaChannel: 1293358091 -> null
+ Done
Canceling job 9297.svc.pads.ci.uchicago.edu
```

No new rlog file was emitted but it did recognize the progress that had been made, the 96 tasks that finished sucessfully above and resumed from 2558 tasks finished.

```
[nbest@login2 files]$ pwd
/home/nbest/bigdata/files
[nbest@login2 files]$
~wilde/swift/src/stable/cog/modules/swift/dist/swift-svn/bin/swift -resume \
> -tc.file tc -sites.file pbs.xml ~/scripts/mcd12q1.swift
> mcd12q1-20100310-1326-ptxe1x1d.0.rlog
Swift svn swift-r3255 (swift modified locally) cog-r2723 (cog modified
locally)
```

```
RunID: 20100311-1027-148caf0a
Progress:
Progress:  uninitialized:4
Progress:  Selecting site:671  Initializing site shared directory:1  Finished
in previous run:1864
Progress:  uninitialized:1  Selecting site:576  Stage in:96  Finished in
previous run:1864
Progress:  Selecting site:1150  Stage in:94  Submitting:2  Finished in
previous run:2558
Progress:  Selecting site:1150  Stage in:94  Submitted:2  Finished in previous
run:2558
Progress:  Selecting site:1150  Stage in:93  Submitting:1  Submitted:2
Finished in previous run:2558
Progress:  Selecting site:1150  Stage in:90  Submitting:1  Submitted:5
Finished in previous run:2558
Progress:  Selecting site:1150  Stage in:90  Submitted:5  Active:1  Finished
in previous run:2558
```

Notes to add (from Mike):

- explain what aspects of a Swift script make it restartable, and which aspects are notrestartable. Eg, if your mappers can return different data at different times, what happens? What other non-determinsitc behavior would cause unpredictable, unexpected, or undesired behavior on resumption?
- explain what changes you can make in the execution environment (eg increasing or reducing CPUs to run on or throttles, etc); fixing tc.data entries, env vars, or apps, etc.
- note that resume will again retry failed app() calls. Explain if the retry count starts over or not.
- explain how to resume after multiple failures and resumes - i.e. if a .rlog is generated on each run, which one should you resume from? Do you have a choice of resuming from any of them, and what happens if you go backwards to an older resume file?
- whap happens when you kill (eg with $^{C)}$ a running swift script? Is the signal caught, and the resume file written out at that point? Or written out all along? (Note case i
  but resume fie was short (54 bbytes) and swift shows no sign of doing a resume? (It silently ignored resume file instead of acknowleging that it found one with not useful resume state in it???) Swift should clearly state that its resuming and what its resume state is.

```
swift -resume ftdock-[id].0.rlog \[rest of the exact command line from initial run\]
```

2.8 Passing an array to swift?

Arrays can be passed to Swift in one of the following ways:

1. You can write the array to a file and read in in swift using readData (or readData2).
2. Direct the array into a file (possibly with a "here document" which expands the array) and then read the file in Swift with readData() or process it with a Swift app() function?
3. You can use @strsplit on a comma separated command line arg and that works well for me.

2.8.1 Mappers

SimpleMapper

```
$ cat swiftapply.swift
```

```
type RFile;
trace("hi 1");
app (RFile result) RunR (RFile rcall)
{
  RunR @rcall @result;
}
trace("hi 2");
RFile rcalls[] ;
RFile results[] ;
trace("start");
foreach c, i in rcalls {
  trace("c",i,@c);
  trace("r",i,@filename(results[i]));
  results[i] = RunR(c);
}
```

```
$ ls calldir resdir
calldir:
rcall.1.Rdata  rcall.2.Rdata  rcall.3.Rdata  rcall.4.Rdata
resdir:
result.1.Rdata result.2.Rdata result.3.Rdata result.4.Rdata
$
```

Notes:

how the .'s match prefix and suffix dont span dirs intervening pattern must be digits these digits become the array indices explain how padding= arg works & helps (including padding=0) figure out and explain differences between simple_mapper and filesys_mapper FIXME: Use the "filesys_mapper" and its "location=" parameter to map the input data from /home/wilde/bigdata/*

Abbreviations for SingleFileMapper Notes:

within <> you can only have a literal string as in <"filename">, not an expression. Someday we will fix this to make <> accept a general expression. you can use @filenames( ) (note: plural) to pull off a list of filenames.

writeData()

example here

```
$ cat writedata.swift
```

```
type file;

file f <"filea">;
file nf <"filenames">;
nf = writeData(@f);
```

```
$ swift writedata.swift
Swift svn swift-r3264 (swift modified locally) cog-r2730 (cog modified
locally)
RunID: 20100319-2002-s9vpo0pe
Progress:
Final status:
$ cat filenames
filea$
```

StructuredRegexpMapper IN PROGRESS This mapper can be used to base the mapped filenames of an output array on the mapped filenames of an existing array. landuse outputfiles[] <structured_regexp_mapper; source=inputfiles, location="./output",match="
transform="\\1histogram">;

Use the undocumented "structured_regexp_mapper" to name the output filenames based on the input filenames:

For example:

```
login2$ ls /home/wilde/bigdata/data/sample
h11v04.histogram  h11v05.histogram  h12v04.histogram  h32v08.histogram
h11v04.tif        h11v05.tif        h12v04.tif        h32v08.tif
login2$

login2$ cat regexp2.swift
type tif;
type mytype;

tif  images[]<filesys_mapper;
location="/home/wilde/bigdata/data/sample", prefix="h", suffix=".tif">;

mytype of[] <structured_regexp_mapper; source=images, match="(h..v..)",
transform="output/myfile.\\1.mytype">;

foreach image, i in images {
   trace(i,@filename(images));
   trace(i,@filename(of[i]));
}
login2$

login1$ swift regexp2.swift
Swift svn swift-r3255 (swift modified locally) cog-r2723 (cog modified
locally)

RunID: 20100310-1105-4okarq08
Progress:
Swift script trace: 1, output/myfile.h11v04.mytype
Swift script trace: 2, home/wilde/bigdata/data/sample/h11v05.tif
Swift script trace: 3, home/wilde/bigdata/data/sample/h12v04.tif
Swift script trace: 0, output/myfile.h32v08.mytype
Swift script trace: 0, home/wilde/bigdata/data/sample/h32v08.tif
Swift script trace: 3, output/myfile.h12v04.mytype
Swift script trace: 1, home/wilde/bigdata/data/sample/h11v04.tif
Swift script trace: 2, output/myfile.h11v05.mytype
Final status:
login1$
```

3 Swift on Computational Infrastructures

3.1 Introduction

We will use the following Swift example, called "catsn" throughout the present section of this cookbook:

```
type file;

app (file o) cat (file i)
{
  cat @i stdout=@o;
}

file out[]<simple_mapper; location="odir", prefix="f.", suffix=".out">;
foreach j in [1:@toint(@arg("n","1"))] {
  file data<"data.txt">;
  out[j] = cat(data);
}
```

3.2 Beagle

Beagle is a University of Chicago's Computational Institute super computer dedicated to bioinformatics research. It is a cray XE6 machine running PBS job scheduler. More on Beagle could be found here.

Swift is available on Beagle as a software module. This recipe will walk you through a very simple example running Swift on Beagle. The prerequisites for running this example on Beagle is an access to and a project allocation on Beagle. The example is a simplest form of a real application which involves reading from a disc file and writing onto another disc file. It runs the `/bin/cat` Unix command to read an input file called `data.txt` and write it to an output file in a folder `outdir`, called `f.nnn.out`.

Swift Example

The steps are as follows:

step 1. Load the Swift module on Beagle commandline as follows: `$ module load swift`

step 2. Create and change to a directory where your Swift related work (source, inputs, outputs) will stay. (say, `$ mkdir swift-lab`, followed by, `$ cd swift-lab`)

step 3. copy the folder at `/home/ketan/labs/catsn` to the above directory. (`$cp -r /home/ketan/catsn .` followed by `$ cd catsn`).

step 4. In the sites file: `sites.xml` make the following two changes:

1. change the path of the `<workdirectory>` element to your preferred location (this location is **required** to be on /lustre, say to `/lustre/beagle/<yourname>/swift-lab/swift.workdir`), and
2. Change the project name to your project (for instance, `CI-CCR000013`) . The workdirectory will contain execution data related to each run, e.g. wrapper scripts, system information, inputs and outputs.

step 5. Run the example using following commandline:

```
$ swift -config cf -tc.file tc -sites.file sites.xml catsn.swift -n=1
```

You can further change the value of `-n` to any arbitrary number to run that many number of concurrent `cat`

step 6. Check the output in the generated `outdir` directory (`ls outdir`)

3.3 PADS

Swift on PADS To execute your Swift script on the PADS cluster use this command:

```
swift -tc.file tc -sites.file pbs.xml catsn.swift
```

where the contents of a simple pbs.xml sites file could be:

```
<config>
  <pool handle="pbs">
    <execution provider="pbs" url="none"/>
    <profile namespace="globus" key="queue">fast</profile>
    <profile namespace="globus" key="maxwalltime">00:05:00</profile>
    <profile namespace="karajan" key="initialScore">10000</profile>
    <profile namespace="karajan" key="jobThrottle">.10</profile>
    <filesystem provider="local"/>
    <workdirectory >/home/you/swiftwork</workdirectory>
  </pool>
</config>
```

3.4 Fusion

Fusion is an Argonne-LCRC computational facility. More information on Fusion can be found here.

In the current section, we present a recipe to use Swift on Fusion system. Fusion employs a PBS schedular. Consequently, we will use the Coasters PBS provider. The jobmanager string will be "local:pbs" as follows:

```
<execution provider="coaster" jobmanager="local:pbs" url="none" />
```

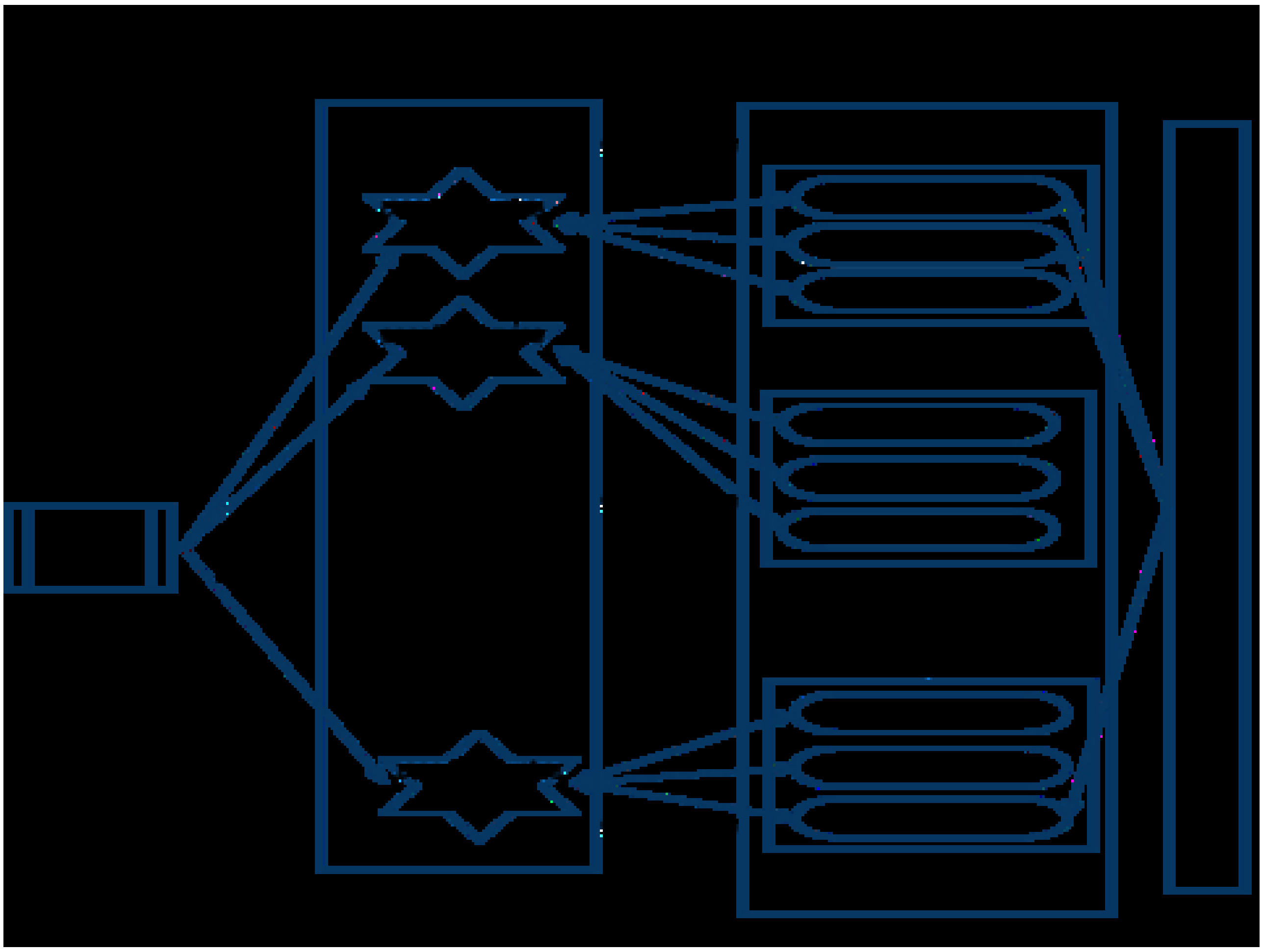
The complete sites.xml file is as follows:

```
<config>
  <pool handle="pbs">
    <execution provider="coaster" jobmanager="local:pbs" url="none" />
    <profile namespace="globus" key="project">ProteinPrediction</profile>
    <profile namespace="globus" key="queue">batch</profile>
    <profile namespace="globus" key="maxtime">1000</profile>
    <profile namespace="globus" key="slots">1</profile>
    <profile namespace="globus" key="nodeGranularity">2</profile>
    <profile namespace="globus" key="maxNodes">2</profile>

    <profile namespace="karajan" key="jobThrottle">0.23</profile>

    <filesystem provider="local"/>
    <workdirectory >/fusion/gpfs/home/${HOME}/SwiftWork</workdirectory>
  </pool>
</config>
```

3.5 OSG

This section describes how to get Swift running on the OSG Grid. We will use a manual coaster setup to get Swift running on OSG.

Coaster setup on OSG The following figure shows an abstract scheme for the manual coasters setup on OSG.



In the following steps, we will go through the process of manually setting

3.6 Bionimbus

This section explains a step by step procedure on getting Swift running on the Bionimbus cloud. We will use the *manual coasters* configuration on the Bionimbus cloud.

step1. Connect to the gateway (ssh gatewayx.lac.uic.edu)

step2. Start a virtual machine (euca-run-instances -n 1 -t m1.small emi-17EB1170)

step3. Start the coaster-service on gateway `coaster-service -port 1984 -localport 35753 -nosec`

step4. Start the Swift-script from the gateway using normal Swift commandline

```
swift -config cf -tc.file tc -sites.file sites.xml yourscript.swift -aparam=999
```

cf

```
wrapperlog.always.transfer=true
sitedir.keep=true
execution.retries=1
lazy.errors=true
status.mode=provider
use.provider.staging=true
provider.staging.pin.swiftfiles=false
foreach.max.threads=100
provenance.log=true
```

tc

```
localhost modftdock /home/ketan/swift-labs/bionimbus-coaster-modftdock/app/modftdock.sh  ↩
    null null GLOBUS::maxwalltime="1:00:00"
```

(See below for a sample sites.xml for this run)

step5. Connect back to gateway from virtual machines using reverse ssh tunneling as follows:

From the gateway prompt `ssh -R *:5000:localhost:5000 root@10.101.8.50 sleep 999`

WHERE: *=network interface, should remain the same on all cases

localhost=the gateway host, should remain the same

5000(LEFT OF localhost)=the port number on localhost to listen to **THIS WILL vary depending upon which port you want to listen to

5000(RIGHT OF localhost)=the port on target host that you want to forward

root@10.101.8.50=the ip of the Virtual Machine on Bionimbus cloud, this will vary based on what ip you get for your Virtual Machine instance

#On anywhere as long as provide the correct callback uri: here the "http://140.221.9.110:42195" is the callback uri of previous ones

step6. Start the worker from the virtual machine `worker.pl http://localhost:42195 tmp /tmp # where 42195 is the port where the coaster service is listening to the workers`

sites.xml for the above run

```
<config>
  <pool handle="localhost">
    <execution provider="coaster-persistent" url="http://localhost:1984" jobmanager="local: ↩
        local"/>
    <profile namespace="globus" key="workerManager">passive</profile>

    <profile namespace="globus" key="workersPerNode">4</profile>
    <profile namespace="globus" key="maxTime">10000</profile>
    <profile namespace="globus" key="lowOverAllocation">100</profile>
    <profile namespace="globus" key="highOverAllocation">100</profile>
    <profile namespace="globus" key="slots">100</profile>
    <profile namespace="globus" key="nodeGranularity">1</profile>
    <profile namespace="globus" key="maxNodes">10</profile>
    <profile namespace="karajan" key="jobThrottle">25.00</profile>
    <profile namespace="karajan" key="initialScore">10000</profile>
    <profile namespace="swift" key="stagingMethod">proxy</profile>
    <filesystem provider="local"/>
    <workdirectory>/home/ketan/swift-labs/bionimbus-coaster-modftdock/swift.workdir</ ↩
        workdirectory>
  </pool>
</config>
```

4 Debugging Swift

Swift errors are logged in several places. In case of a failed Swift run, following places should be inspected for clues:

1. Text from standard output and standard error produced by running Swift
2. The .log file from this run. It will be named script.uniqueID.log where "script" is the name of your Swift script source file, and uniqueID is a long unique id which starts with the date and time you ran the swift command.

3. $HOME/.globus/scripts directory on the host on which you run the Swift command, when swift is submitting to a local scheduler (Condor, PBS, SGE, Cobalt)
4. $HOME/.globus/coasters directory on remote machines on which you are running coasters
5. $HOME/.globus/* files on remote systems accessed via Coaster providers

5 Log Processing

To properly generate log plots, you must enable VDL/Karajan logging. Make sure log4j.properties contains:

```
log4j.logger.swift=DEBUG
log4j.logger.org.globus.cog.abstraction.coaster.service.job.manager.Cpu=DEBUG
```

5.1 Make a basic load plot from Coasters Cpu log lines

Generate the log (may set log4j.logger.swift=INFO for this one) (assuming the log is titled swift-run.log)

Convert the log times to Unix time

```
./iso-to-secs < swift-run.log > swift-run.time
```

Make the start time file (this contains the earliest timestamp)

```
make LOG=swift-run.log start-time.tmp
```

or

```
extract-start-time swift-run.log > start-time.tmp
```

Normalize the transition times

```
./normalise-event-start-time < swift-run.time > swift-run.norm
```

Build up a load data file:

```
./cpu-job-load.pl < swift-run.norm > load.data
```

Plot with the JFreeChart-based plotter in usertools/plotter:

```
lines.zsh load.cfg load.eps load.data
```

5.1.1 Make a basic job completion plot from Coasters Cpu log lines

Same as above, but, build up a completed data file:

```
./cpu-job-completed.pl < swift-run.norm > completed.data
```

Plot with the JFreeChart-based plotter in usertools/plotter:

```
lines.zsh completed.cfg completed.eps completed.data
```

5.1.2 Make a basic Block allocation plot from Coasters Block log lines

Same as above, but:

Build up a block allocation data file:

```
./block-level.pl < swift-run.norm > blocks.data
```

Plot with the JFreeChart-based plotter in usertools/plotter:

```
lines.zsh blocks.{cfg,eps,data}
```

5.2 Problem Reporting

When reporting problems to swift-user@ci.uchicago.edu, please attach the following files and information:

1. tc.data and sites.xml (or whatever you named these files)
2. your .swift source file and any .swift files it imports
3. any external mapper scripts called by your .swift script
4. all text from standard output and standard error produced by running the swift command
5. The .log file from this run. It will be named swiftscript.uniqueID.log
6. where "swiftscript" is the name of your *.swift script source file, and uniqueID is a long unique id which starts with the date and time you ran the swift command.
7. The swift command line you invoked
8. Any swift.properties entries you over-rode ($HOME/.swift/swift.properties, -config.file argument properties file, any changes to etc/swift.proerties from your swift distribution)
9. Which swift distribution you are running (release; svn revisions; other local changes you mave have made or included)