

# Scaling Swift

Ben Clifford

[benc@ci.uchicago.edu](mailto:benc@ci.uchicago.edu)

<http://www.ci.uchicago.edu/swift/>

# Running lots of jobs with Swift

- Introduction to Swift:
  - Swift is a parallel scripting language and engine to execute it.
  - Sits on top of other components – from Globus, and below that, LRMs and filesystems on compute resources
  - Trying to hide people from the complexity of these components.
- Some problems and solutions specifically related to scaling the number of jobs – quick tour of several different issues – neither in depth, nor comprehensive.
- Not intended as a criticism of any particular component that I talk about – they are examples.

# Where Swift fits in the landscape

Application (eg fmri)  
**THIS IS THE POINT**  
every thing else below is somewhat incidental



Swift / SwiftScript



GridFTP

Clusters

GRAM

Falkon

PBS

GPFS

# Describing $10^6$ jobs

- Science user doesn't start with  $10^6$  jobs in mind. Start with some high level application that it happens to be convenient to describe as  $10^6$  jobs.
- Describe that application in SwiftScript.
- Our approach is given that application, break it into “application procedures”; each “application procedure” is:
  - separately schedulable
  - restartable, relocatable - unit of checkpointing and fault tolerance
  - communicates with other “application procedures” only through write-once files
- Describe how broken up chunks are connected using SwiftScript language; write each chunk in some other language – app specific. C, R, PERL.

# Hello world

```
type messagefile;
```

```
app (messagefile f) greeting() {  
    echo "Hello, world!" stdout=@f;  
}
```

```
messagefile outfile <"hello.txt">;
```

```
outfile = greeting();
```

# Parameter sweep

```
type file;
```

```
app (file min) mxmodel_processor (file cov_matrix, file dot_r, int  
    numcol, int modnum){
```

```
    Rinvoke @dot_r @cov_matrix numcol modnum;
```

```
}
```

```
int totalperms[] = [0:65534];
```

```
file modmin[] <simple_mapper;dir="results/">;
```

```
foreach perm,i in totalperms {
```

```
    modmin[i] = mxmodel_processor(covmx, cov_script, numcol,  
    perm);
```

```
}
```

- Can users submit millions of jobs simply by using a template from which jobs are created and submitted automatically?
- [q3.1](#)

# SwiftScript language scalability

- Mappers+foreach = good for large datasets.
- Once app is described for a few data files, description doesn't change for larger number of data files.
- As the language becomes more convenient for describing pieces of an application, people start describing things differently. 5y ago the Globus story was “submit big jobs with big inputs and outputs”. Now people describe jobs that are a few seconds long with data files that are only a few kilobytes – want to execute things differently.

# swift execution model

SwiftScript procedure call

Site Selection

rate limiting

remote file access

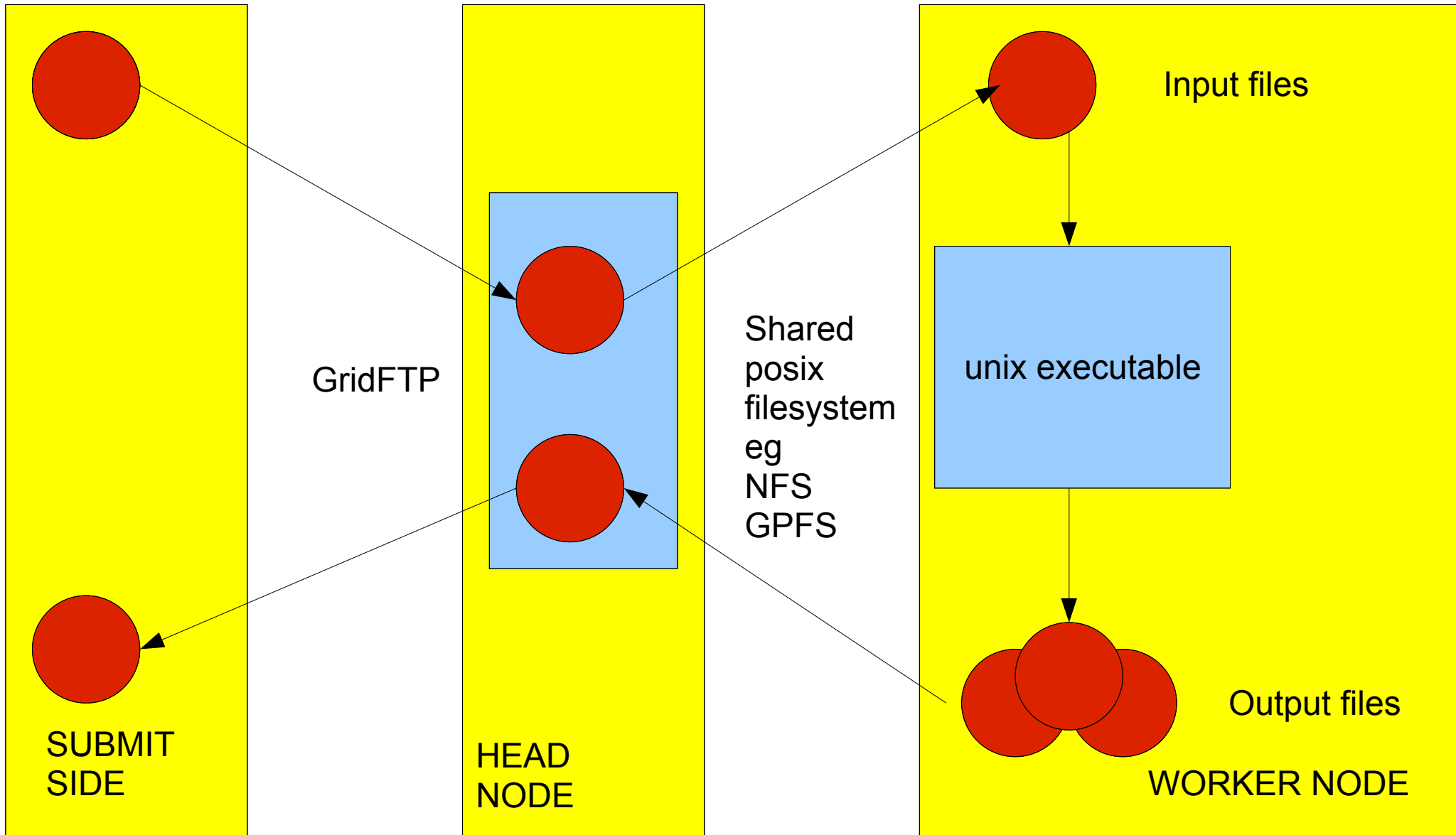
remote job execution

site file cache

worker node wrapper

user executable

# Swift vs shared filesystems

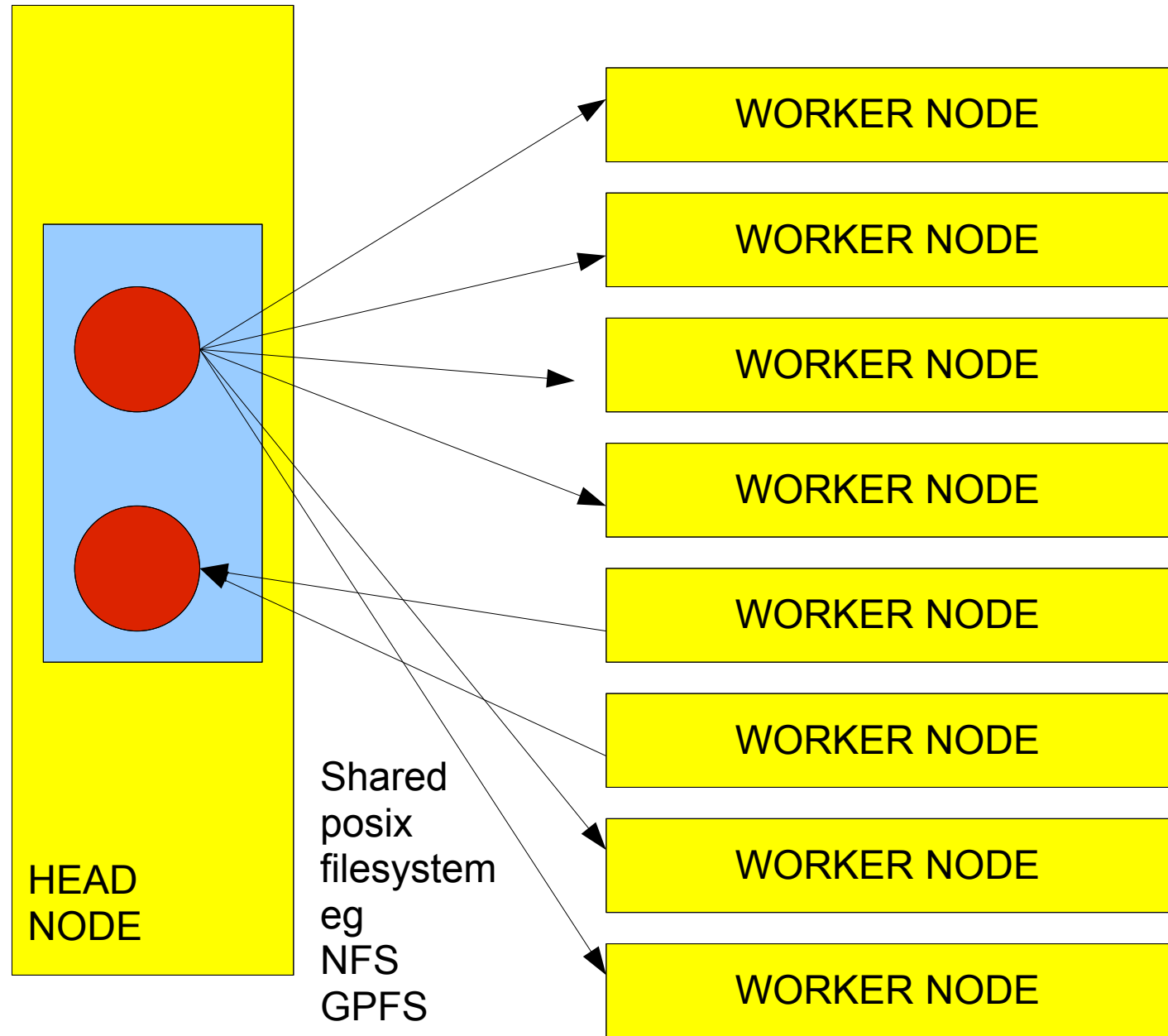


# Swift vs shared filesystems

NFS – bandwidth limitations on single NFS server

GPFS – much more scalable, but...

Original Swift filesystem layout was perfectly tuned to cause terrible GPFS lock contention – eg under load, 30s to touch a status file or write a log line.



# File abstraction benefits and costs

- File abstraction – application executable starts running in some directory, with the input files there; and it leaves the output files in the same directory.
- Can change where 'some directory' is
  - change layout to be GPFS friendly (hierarchy)
  - move to a local RAM disk for better posix performance
- Can change how those files get there
  - this model doesn't need a shared fs. other ways with different benefits and costs, eg pull gridftp
- No change to SwiftScript or application executables.

# Swift execution model

- Application expects to start running in some directory, with input files in place.
- Launch using:
  - GRAM2, GRAM4, Falkon, CoG coasters, ...
- None best on all axes – GRAM2 widely deployed but scales very poorly. GRAM4 scales in job numbers much much better, but much less deployed.
- Target space has shifted from hour long jobs to seconds-long jobs – GRAM + most LRMs deal very badly with that.
- Provisioning code eg Falkon performs **much** better, but codebase less mature.
- Abstraction makes it relatively straightforward to plug in different execution mechanisms.

# Falkon integration

- Relatively easy to plug Falkon into Swift.
- But difficult to get performance numbers on same scale as “pure” Falkon
- Perception that Swift is “just running a bunch of jobs” is wrong.
- Falkon gives us good execution performance, so now we are data-bound.
- Before provisioning, we were compute-bound (or job submission system bound)
- Integration of pieces is hard – at least to get decent performance – despite wonderful scalability numbers for each piece.

# Rate limiting

- If you have a million jobs, can't send a million GRAM jobs or a million GridFTP transfers all at once. So you have rate limiting.
- Swift rate limiting gives each site a score. Score controls number of transfers/executions on that site at any one time.
- Success = score up. Failure = score down
- Very simple to understand. Usually works quite well.
- Some problems:
  - When a GRAM2 site is 'overloaded' (in the opinion of a site admin – eg if head node loadavg is 300) it still takes jobs – they don't 'fail'. So how does Swift know this site is overloaded? It would be nice to have some more feedback from the site [Q2.3](#)
  - Users change throttles to get more performance. This can be a hammer of death if used without understanding or with too much hope about what the underlying services can support.
- Lots of graphing and stats on Swift runs. But decent understanding of how the whole system works is needed to know what, if anything, needs tweaking; contrary to goal of making this work for a user without them needing to understand all the guts.
- What is the “right” performance level anyway?

# multisite

- Our background is Grid – so initial assumption is that people want to run multisite.
- What is the future here? Multisite? single site? blurred/hybrid
- Should we work on improving multi-site case or on single-site case?
  - getting app executables running on a new site is hard for users – it is software packaging and deployment, which people get employed full time to do
  - Site affinity and other site selection issues. Lots of research - “build a better site selector” seems to be
  - Which sites are available and how are they configured? OSG seems reluctant to have a single Official Information System. In the absence of an information system, it turns out that we do site status detection fairly well by attempting to use and letting broken sites get heavily rate limited. [Q2.3](#)
- Multisite troubles push people towards running on a single site with a large number of CPUs (eg one large TG sites rather than many smaller OSG sites).

# System complexity

- Difficult to understand the whole system.
- Lack of understanding leads to mistakenly perceived errors due to surprising behaviour
  - Q: “I see many jobs completing in the PBS queue on remote site, but I do not see the output files appearing on the local side – therefore, stageout is broken”.
  - A: stageout is deferred because we have a rate-limited file staging, and we are preferring to stage in files for new jobs rather than stage out outputs of existing jobs. as soon as all of those are staged in, your output files will rapidly start appearing.
  - Do we prefer to be less surprising but much less performant? Sometimes yes. (debugging vs production)
- Performance tuning – many parameters. needs more knowledge than is desirable.
  - related project: CEDPS logging/debugging

# more info

- demo in Argonne booth Wednesday 3pm
- <http://www.ci.uchicago.edu/swift>
- [benc@ci.uchicago.edu](mailto:benc@ci.uchicago.edu)