

Challenges in grid workflows

Ben Clifford

Computation Institute, University of Chicago

benc@ci.uchicago.edu

about me

- Ben Clifford
- programmer of globus-related stuff since 2002
 - earlier Globus Monitoring and Discovery System (MDS), at ISI in Los Angeles
 - now Workflow at University of Chicago Computation Institute
- present position deals with all layers of running grid stuff
- Mathematician by formal education, programmer/admin/hacker/work
- benc@ci.uchicago.edu
- <http://www.hawaga.org.uk/ben/>

Swift

- Motivation for this is work I've done on and around a software project called Swift.
- This is not a Swift marketing talk.
- However, SWIFT IS GREAT!
- <http://www.ci.uchicago.edu/swift/>

Swift

- Grid workflow system
- Coarse grained grid-scale parallel scripting language (SwiftScript)
- runtime engine to execute SwiftScripts

history of Swift

- VDS – Virtual Data System
- Spawned two on-going projects:
 - Swift – focus on language. Mike Wilde, Argonne/UC
 - Pegasus – focus on planning. Ewa Deelman, ISI

grid scale federation

- Grid technology federate lots of resources.
- In the process, it federates lots of problems for you too
- Which we then encounter
- And then try to solve

topics

- application decomposition
- language expressiveness
- site selection
- submitting your job for execution
- stuff that happens inside a site
- provenance
- distributed debugging

WARNING: Its not all bleak

- Lots of this presentation talks about problems.
- But its not all bad! Many problems solved. Many problems have reasonable proposed solutions.
- Don't leave this seminar thinking everything is broken – lots of stuff works well, but that's not what I'm focusing on here.

application decomposition

- Applications need to split into unix-executable components that communicate through files
- SwiftScript describes
 - the application interfaces – inputs, outputs, paths
 - how the pieces fit together
- lots of people already have applications like this: shell pipelines that can be straightforwardly translated into SwiftScript

application decomposition

- there are other notions of:
 - coupling: MPI
 - component: web services (eg. Cancer Biomedical Informatics Grid (caBIG))
 - data: some computational biology store data in SQL databases or single large files
- There are grid tools for dealing with this...
 - e.g. BPEL, OGSA-DAI
- ... but how do these fit into Swift?
- Do we even want to make them fit?

application installation

- hassle to do on multiple sites
- not easily automated
- research code especially unportable
- amongst the users I see, it is one of the biggest impediments to using multiple grid sites.
- packaging is hard. people have been doing it for many years. it doesn't get much easier. it is a skilled and labour intensive process. solution: skilled manpower (either get someone with the skills or have to spend the time waiting for them to become skilled)

SwiftScript in 10 seconds: invoking echo in swift

```
type messagefile;
```

```
(messagefile t) greeting() {
```

```
    app {
```

```
        echo "hello" stdout=@filename(t);
```

```
    }
```

```
}
```

```
messagefile f <"out.txt">;
```

```
f = greeting();
```

dataset mapping

The contents of variable f live in out.txt:

```
messagefile f <"out.txt">;
```

The contents of array variable a are in several files, names input*.jpeg:

```
jpeg a[] <simple_mapper;  
  prefix="input",  
  suffix=".jpeg">;
```

dataset mapping

jpeg a[]

```
<simple_mapper;prefix="input",suffix=".jpeg">;
```

a[1]	→	input0001.jpeg
a[2]	→	input0002.jpeg
a[3]	→	input0003.jpeg
a[4]	→	input0004.jpeg
a[5]	→	input0005.jpeg

brains.txt

a[6]

dataset mapping

- Swift comes with ~10 mappers
- Application specific mappers can be written to handle different dataset layouts
- People do that quite rarely - existing mappers seem to cover most cases; or mappers are too hard to write compared to arranging data?

The inexorable slide to being a real programming language

- VDL1 was not very expressive:
 - one line = one statement = one execution
 - external generation of VDL1 programs – (very) big VDL1 source files
- “We don't want a real programming language, we just want...”
 - structures and arrays
 - iteration over arrays
 - literals
 - numerical operations (+ - * / %)
 - if() statements
 - strcat
 - ...
- BZZZT!

Programming languages that are not FORTRAN are hard for people to understand

- Dataflow mode / functional style easy to explain for simple cases but but sequential thinking is hard to escape
 - “i want to run an program against every file in a dataset, so first we process index 1 then index 2 then index 3 then index 4... as some kind of for/while loop”
 - vs. “I will use parallel application” rather than expressing the higher level 'map this over that' syntax”
- Similar problems arise when trying to teach functional programming, or other kinds of parallel or concurrent programming (even threads in Java, which are “easy”)
- Users introduce side effects (verboten!)
 - breaks abstraction – retries, multi-site execution, so your prototype code works but it doesn't in production

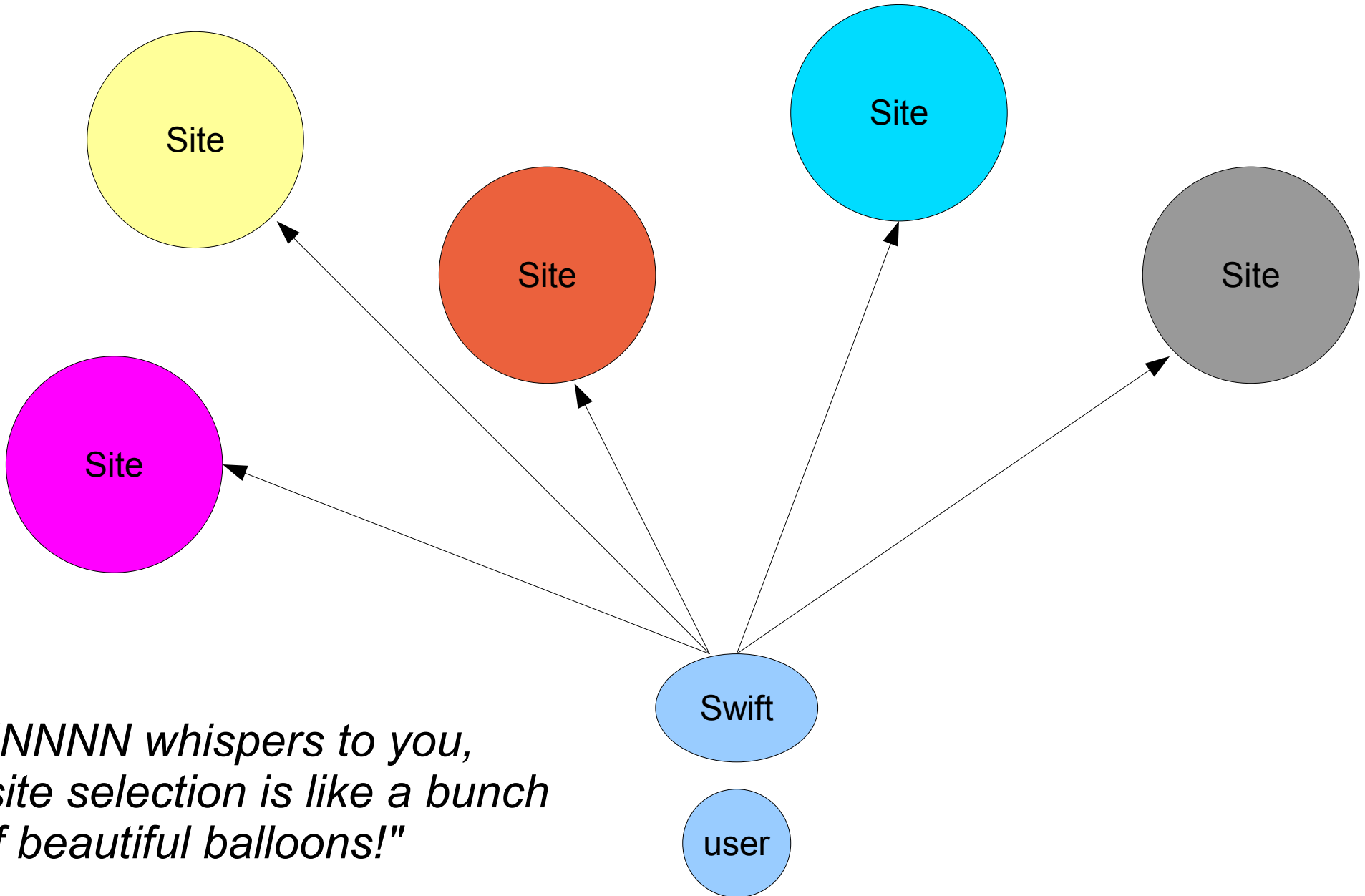
Dataflow/functional language abstraction worth it?

- Yes!
- By constraining expressiveness, we get big benefits
- Portability between sites
- Fault tolerance through restarts – checkpointing is very easy at the course grained level
- Files can be placed anywhere on the remote system (see local filesystems later)
- Hopefully soon, run the same job in multiple locations (see site selection later)
- Users hacking side-effects in drives new functionality

Execution

- That was the language side of things.
- Runtime takes SwiftScript program and generates a list of jobs to run 'on the grid'
- For each job:
 - Site selection
 - Running on a site

site selection

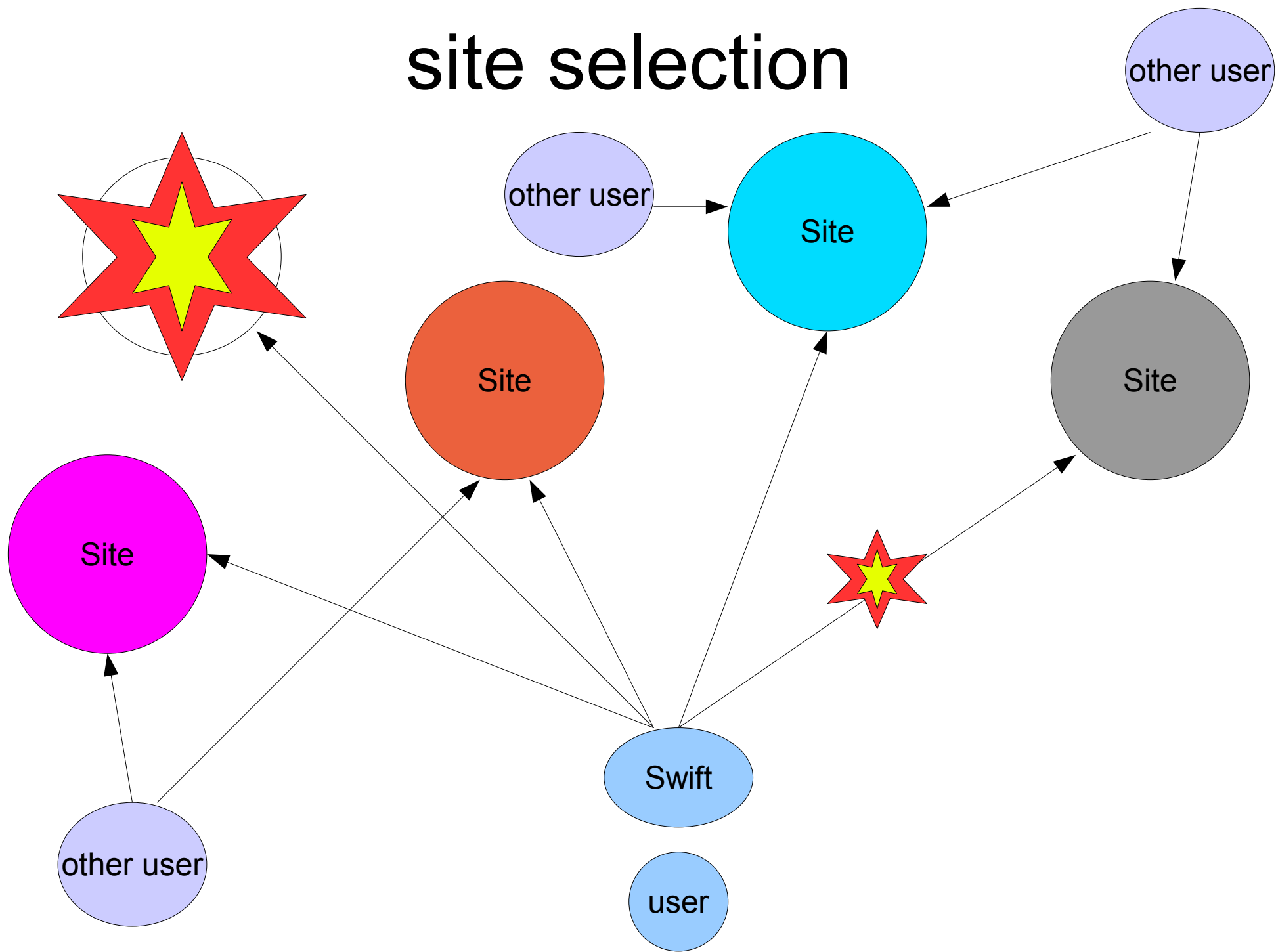


*NNNNN whispers to you,
"site selection is like a bunch
of beautiful balloons!"*

Site selection

- Grid is split into sites.
- Run jobs on a site
- Site selection is picking the best site to run a job on

site selection



site selection

- In Swift, site selection and rate limiting on a site are closely tied together.
- swift has a scoring approach: success +, fail or slow -
- score determines how many task slots a site will get
- easy for users to understand “this site has five slots” and relate that to how much activity they see on a site
-
- deals poorly with transient spikes – if we have 100 jobs on a site, and some transient error kills them all, we tend to overpunish that site compared to what a human would do there.

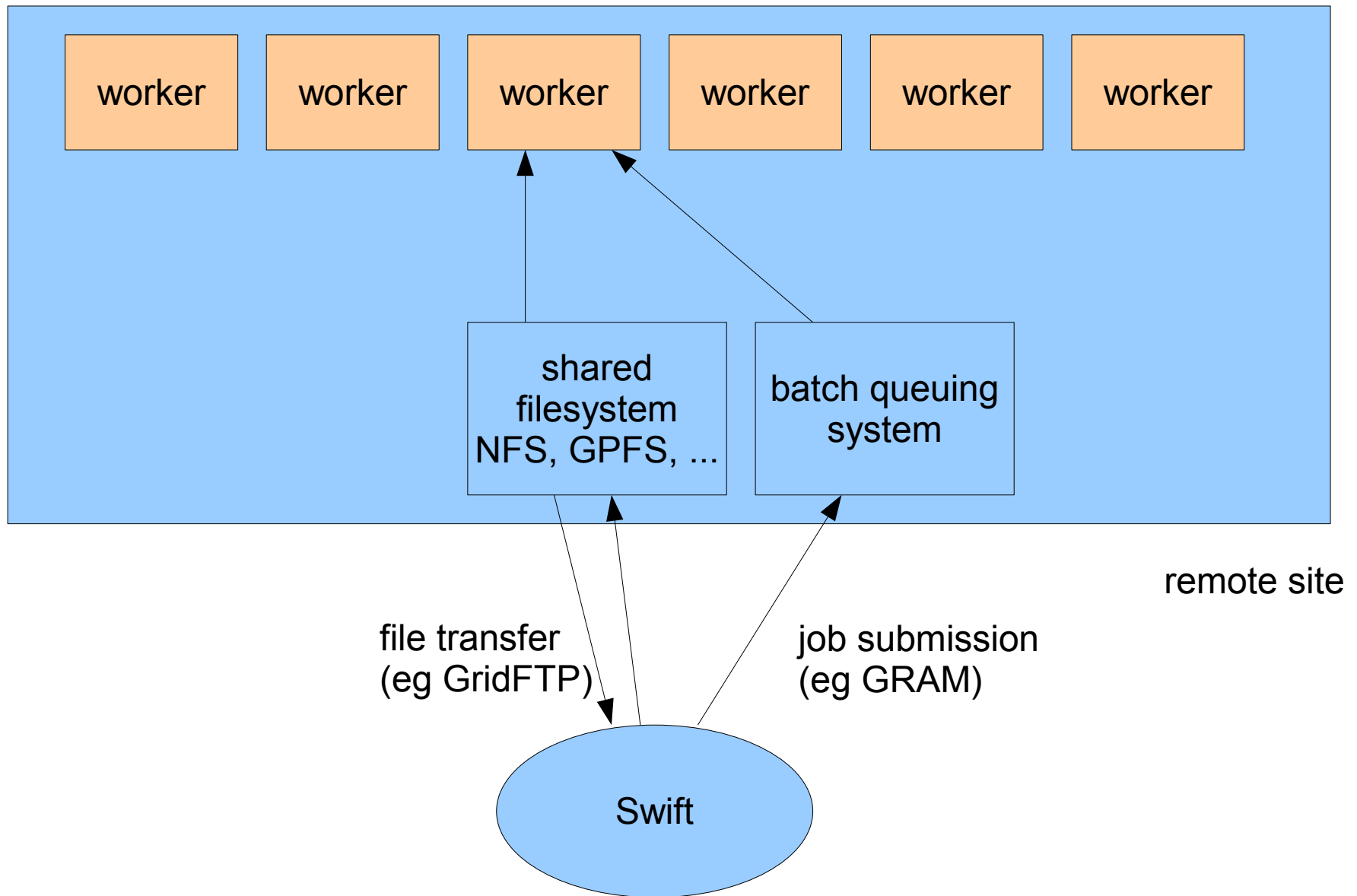
site selection

- must submit something to a site to change scores. so must submit to bad sites in order to detect they are bad, and keep submitting in order to detect if/when they stop being bad.
- if we let a score hit 0, then we will never run against it again, and so it will never get a positive score again – its gone forever (where forever = duration of workflow run)
- if we constrain sites to always have one job slot, then fast-fail failure modes can eat the whole workflow (including retries) very quickly.
- a site might slow-fail – accept a job and never do anything with it (or take a long time (relative to other sites)).
- needed feature: site unselection
 - need to i) realise we've sent a job to a site that isn't fast enough (perhaps totally broken, perhaps just very slow indeed); ii) have the internal infrastructure to unsubmit / resubmit elsewhere.

site selection

- We need information, some of it from the future.
- Grid information systems give a bunch of data about sites, but hard to use it to generate meaningful scores ahead of time.
- Hard to even get a decent upper bound on the expected queue time for a job – users actively discouraged from accurate walltime predictions: excessive punishment for underestimation; tragedy of the commons
- UC has visiting researcher working on some of this (□□)
- Can we use and how can we use what other people have done already? (eg. within Globus, Gridway metascheduler)

running a job on a site



running a job on a site

- Go fast:
 - Get the files to and from the workers
 - Get the workers doing application work
- Go slow:
 - Rate limiting – in Swift, same mechanism as site selection – site 'slots' controlled by both site reliability and manual throttles
- Contradiction causes regular explosions and gridquakes

Get the workers doing application work

- Much more choice (through necessity)
- traditional globus grid mechanisms:
 - GRAM2 (much more deployed on eg. OSG)
 - GRAM4 (works much better than GRAM2)
 - releasing new software is not enough – it takes time (much time) for released services to appear on production resources
 - PBS direct submission – running on head node of cluster
 - separation of resource provisioning and job submission
- Some not-very-supported / experimental ones
 - ssh
 - someone is experimenting with a BOINC provider, for example (another seminar yesterday, but far from here)

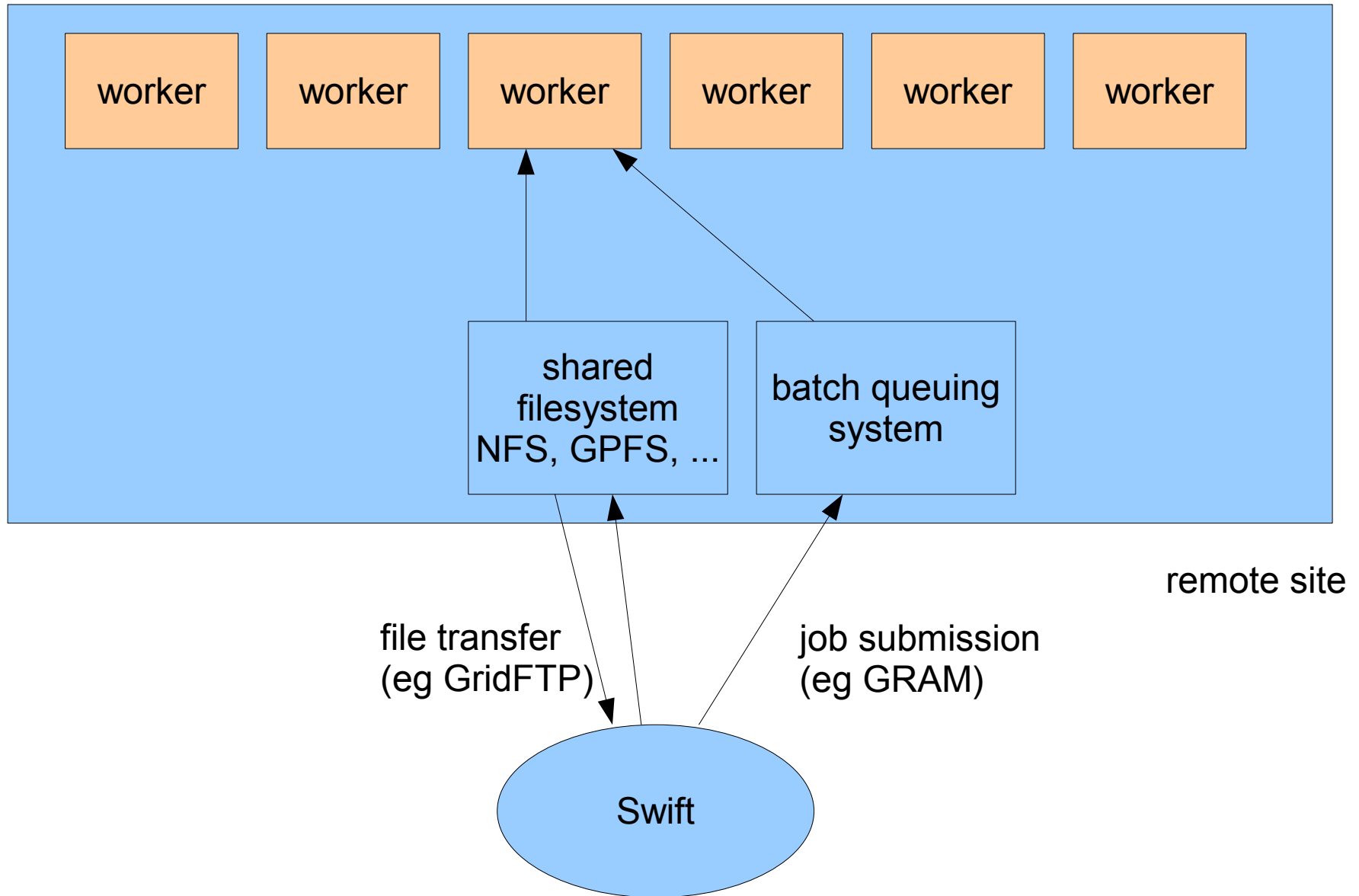
separating provisioning and execution

- Separation of provisioning and job submission
- Provisioning acquires nodes, using GRAM or PBS or ...
 - you own the nodes for long time (~hours)
 - similar to but different from making reservation on site
- Nodes run workers which pull only your jobs
- You can (almost) fill your maxwalltime on a worker node even though you do not know what you want the worker to do at submission time

separating provisioning and execution

- Falcon - research project
 - 3750 trivial tasks/second on 200 CPUs on University of Chicago TeraGrid
 - 3188 trivial tasks/second on 5760 CPUs on SciCortex
 - (without security – polite cough)
 - much faster than the space in which Swift operates
 - <http://people.cs.uchicago.edu/~iraicu/projects/Falcon/>
- CoG coasters – production-oriented code, implementation in progress

running a job on a site



Get files to and from the workers

- File transfer - usually GridFTP (but pluggable)
 - we have made improvements in the Java CoG implementation of GridFTP client as we encounter problems – eg. improved scalability with data channel reuse
- For workflows with lots of very small input files and very high job throughput, some scalability problems somewhere in Swift runtime.
- Shared filesystem scalability problems...

the site-shared filesystem

- site model requires shared posix-like filesystem – shared between worker nodes and some transfer endpoint (eg GridFTP server)
- NFS, GPFS, PVFS
- shared filesystem gives scalability problems
 - hundreds of works all hitting the same shared filesystem space
- different filesystems scale differently.
 - NFS – shared space on one server. not good to have hundreds of nodes trying to transfer to/from same server
 - GPFS scales better in total throughput, but perfoms really really badly when one file/directory accessed from multiple workers (eg 30s to write one line to a log file shared between all workers). Can rearrange data files so that they are in a hierarchical structure.
- application file access patterns affect execution time. sometimes it is much quicker to copy all input files to worker node local fs at start of execution rather than allow applications to access the shared fs.

removing the site-shared filesystem

- get rid of shared filesystem
- provisioning system is also in charge of getting files to the appropriate worker node, without using posix shared filesystem
- substantially more complex worker-side code
- SwiftScript programs shouldn't see any difference – advantage of strong abstraction
- beginning to work on this as part of falkon research
- exciting new bottlenecks to look forward to (probably internal cluster network)

distributed debugging

- something is failing (runs do not complete)
- something is not going fast enough

something is failing

- Workflow is failing with a 'workflow failed' error
- Propagating error messages from down below to something that makes sense for a higher level user
- Log unification

Understanding errors from high in the stack

- many different components, each with their own peculiar failure modes and error reporting mechanisms
- lower level errors don't make sense higher up:
 - walltime violation with PBS+GRAM2 manifests as job mysteriously disappearing and being reported as 'Completed' rather than 'Failed'. Subsequently we get a GridFTP error 'status file /workdir/a/b/111111 not found'. We successfully detect something went wrong – but it doesn't look like a walltime violation
- component level errors change depending on underlying components, even though Swift is still the same version:
 - swap gram2 to gram4 - 'GRAM Error Code 7' becomes something else
- components are sometimes badly behaved in their error reporting
 - GRAM2 doesn't return application exit codes; PBS walltime violations come via email.

Log Unification

- logs in many places (on each worker node and on each infrastructure host)
- logs in many different formats
- hard to gather. sometimes not accessible to mortals (eg PBS logs)
- hard to correlate – each layer uses different IDs
- CEDPS is working (in part) on that, but their work will not be deployed for a while and will not cover everything in the stack.
<http://www.cedps.net>
- I have a bunch of hacky code that sits alongside Swift proper for parsing logs and giving different views of – textual, graphical

something is not going fast enough

- workflow completes OK, but “it isn't fast enough”
- what is fast enough? lots gut feeling involved.
- vain hope for speedup = $O(\text{number of CPUs in whole grid})$
- explosion of tweakables – many configuration parameters to understand and guess appropriate values for.
- same workflow with different input files or different environment or different day can have different scalability limits - “the problem is job submission is slow...” move to faster job submission “the problem is file access in is slow...”

Whole system understanding

- needs a decent understanding of all the pieces to not be led astray. can full time job doing that, let alone actually working on the applications themselves.
- Q: Jobs are not being submitting fast even though I set the job throttle high
- A: Input data stage-in is a bottle neck – jobs can't be submitted until their input data is there.
- Q: I see jobs completing fast enough but most of their output files aren't appearing – so they must be failing.
- A: if lots of files are staging in for other jobs still, then stageouts may be way down in the queue. what often happens is that almost no stageout happens until very near the end of a workflow, and then bang! all happens very quickly towards the end

I love graphs.

- Some of the most useful scalability debugging information I get from graphing whole runs.
- But still needs understanding of whats going on inside to know what the graph should look like.

provenance

- what is provenance? (seem to lose users at the first hurdle, even though they often end up asking for provenance using different words)
- it is very easy to collect a bunch of data and throw it into some database.
- much harder to make that database:
 - easy to query
 - quick to query

provenance

- flexibility on the part of schema designers pushes schema design into the future and onto query implementors. oh so tempting.
- “inner platform effect” antipattern - thedailywtf.com
 - trying to make a provenance database that isolates user from the complexities of SQL whilst providing a platform that is similarly powerful to SQL; end up making a poor reimplementaion of SQL.

provenance

- traditional schema/query languages have poor support for (essentials?):
 - transitive relations
 - differing levels of trust in database entries (different from read/write permissions)
- much of this is database design stuff, nothing to do per-se with provenance

provenance

- VDS1 had VDC. Was in first provenance challenge, and did well in terms of query completion;
- Swift Provenance database a component under development.
- Lots of experimenting with querying – what languages to use? how to expose sensibly to users?
- (sql, xquery, sparql, graphgrep, prolog)
- Strong relation to semantic web – is part of?

summary

All of these:

application decomposition

language expressiveness

site selection

submitting to a site

stuff that happens inside a site

provenance

distributed debugging

are things you have to keep in your head to run an application – plus details of your actual application.

Most are not Swift specific - any non-trivial use of the grid is going to encounter lots of this stuff.

WARNING: Its not all bleak

- Lots of this presentation talks about problems.
- But its not all bleak. Many problems solved. Many problems have reasonable proposed solutions.
- Don't leave this seminar thinking everything is broken – lots of stuff works well, but that's not what I'm focusing on here.

fin