Computation Institute

SkeletonKey: Simplifying Data and Software Access for Users

Suchandra Thapa
Computation Institute / University of Chicago

March 12, 2013

Introduction

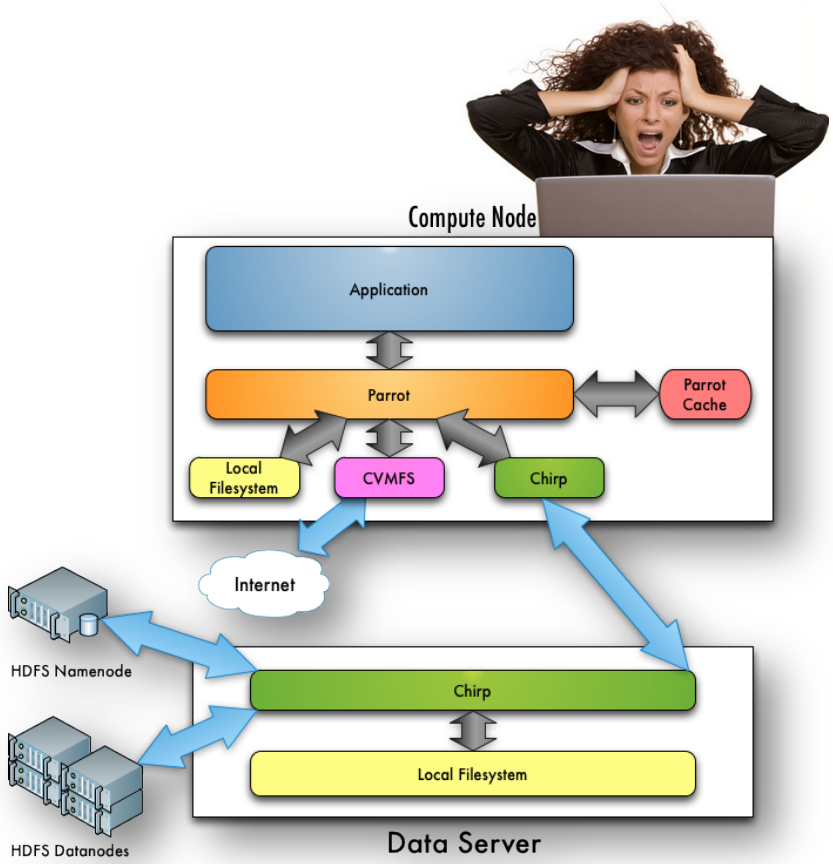
- What is SkeletonKey?
- How is it used?
- Performance Comparisons
- Conclusion

Simplifying Software and Data Access

- A general issue facing Campus Infrastructure providers is simplifying user access to software and data from heterogeneous resources, both on and off campus
- Good results at UW-GLOW using the Parrot tool as job wrapper and a Chirp proxy service for data and software
- Our goal is leverage good solutions, and add value by further simplifying their use for campus communities
- Specifically want a user-focused tool that encapsulates complexity so that users get started quickly
  - **SkeletonKey** does this by creating configurable user job wrappers to correctly use Parrot and Chirp for a given Campus and/or VO context

Simplifying

Directly

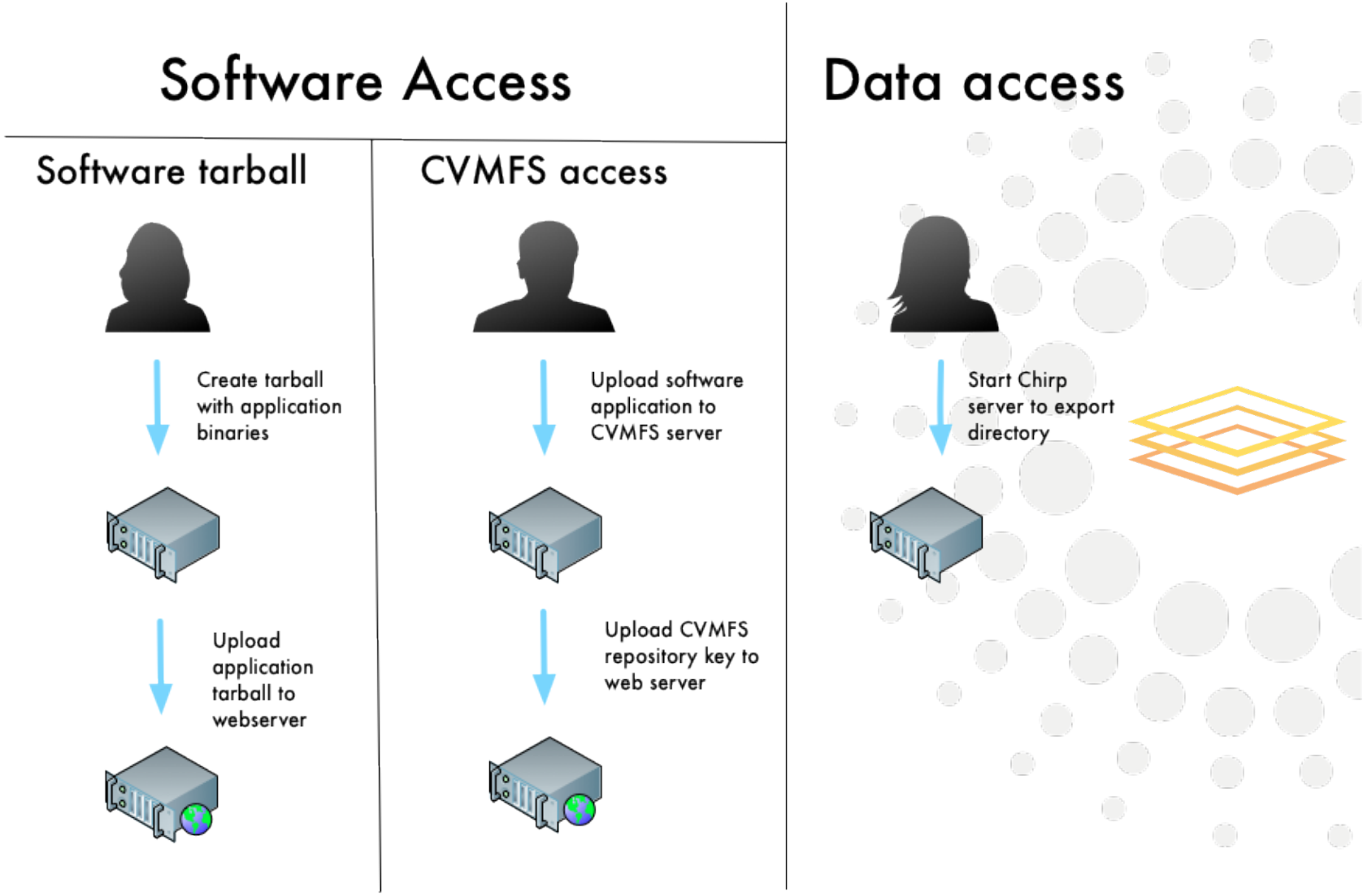


With SkeletonKey

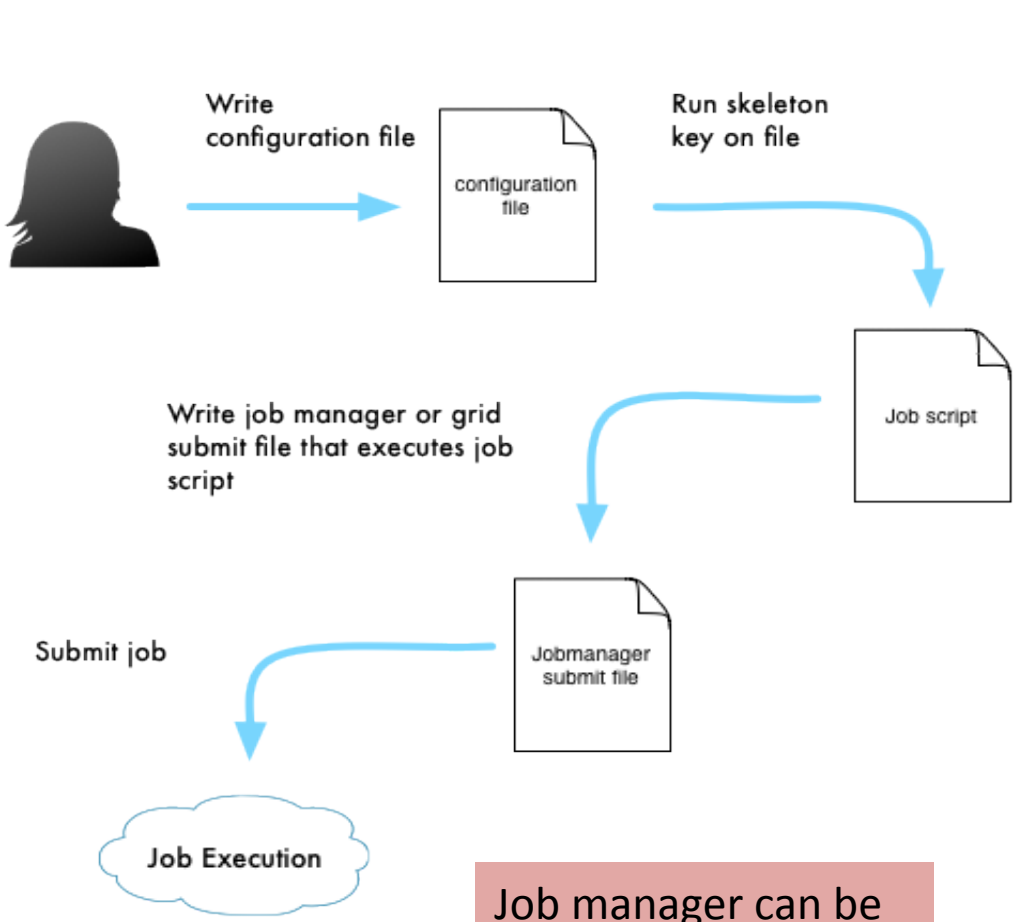
How SkeletonKey works (simple!)

- A wrapper function is specified with an easy to understand configuration file
- SkeletonKey generates a script from the configuration file to be used in a job manager and/or grid submit file
  - a portable file that can be copied to another system and run

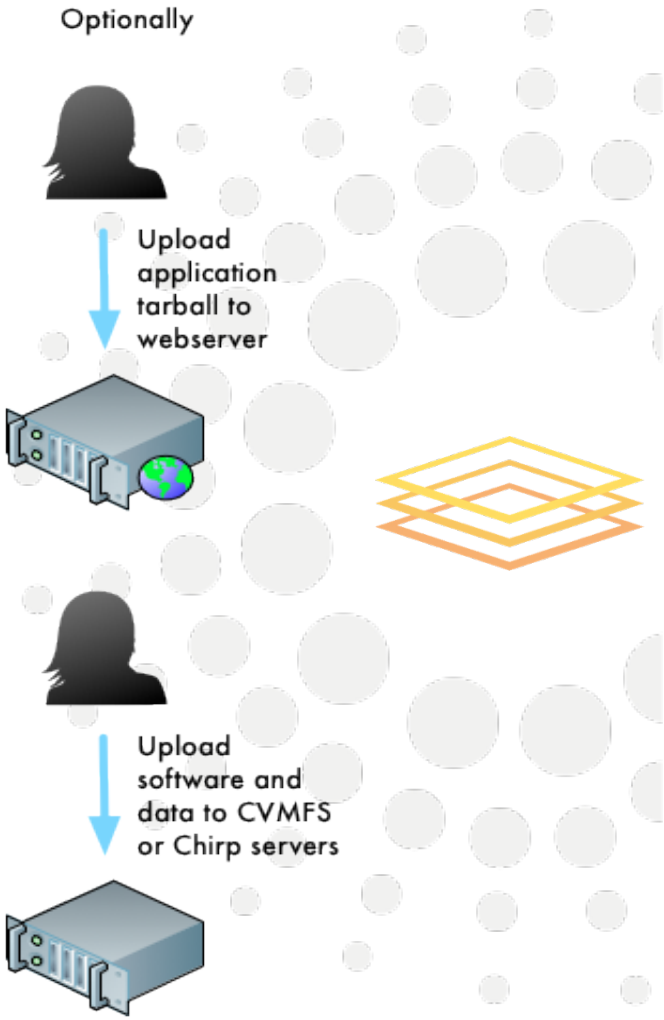
Preparations for software or data access



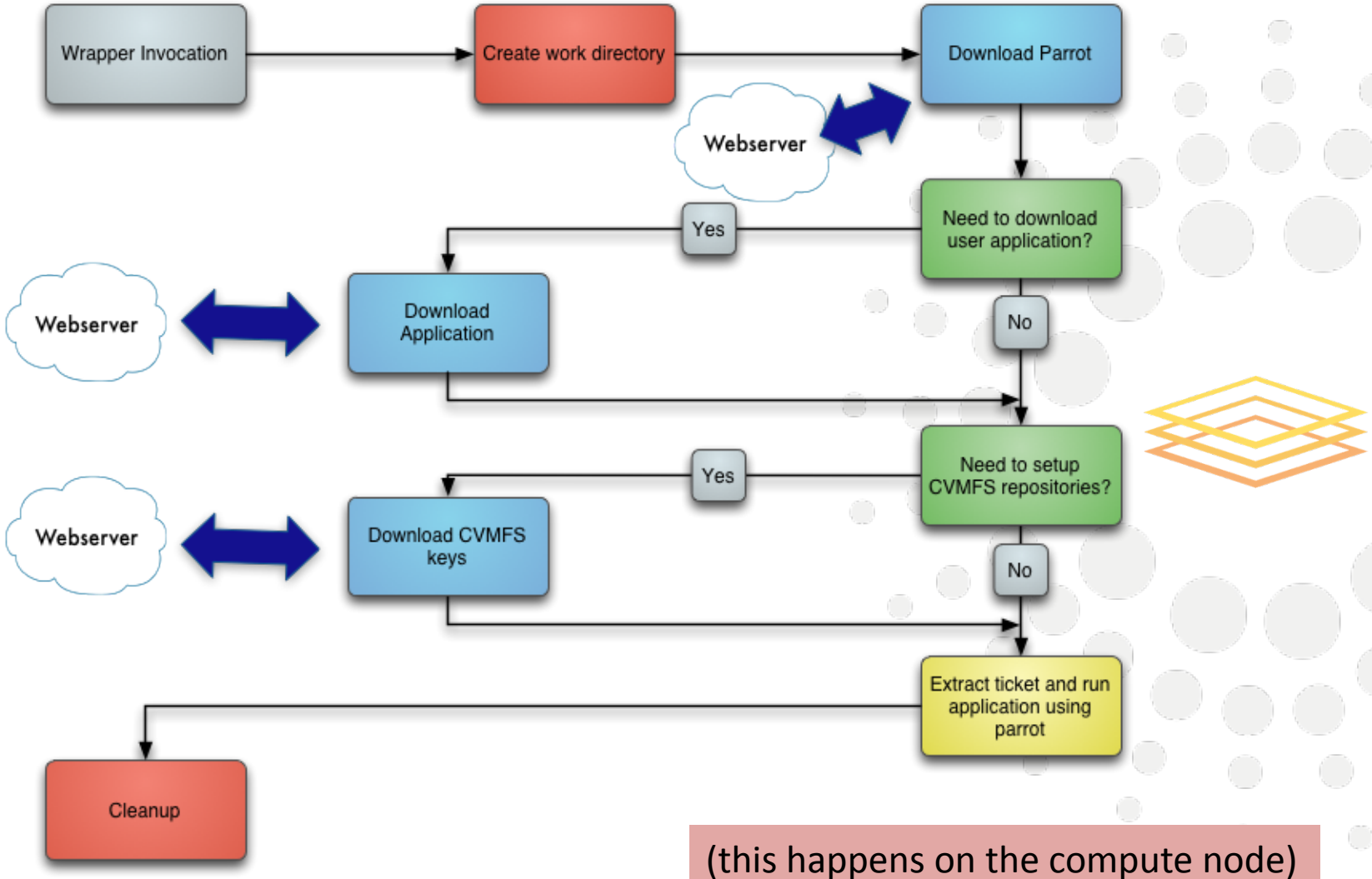
Typical SkeletonKey workflow



Job manager can be HTCondor, PBS, SGE, LSF, HTCondor-G, etc.



Internal workflow on compute node of the generated job wrapper



(this happens on the compute node)

SkeletonKey configuration

- Uses a ini configuration file
  - Widely used so it is probably familiar to users
- Has four sections
  - `Application` section to specify where and how to run user application
  - `Directories` section to specify data access
  - `CVMFS` section to specify access to CVMFS repositories
  - `Parrot` section to allow users to specify specific versions of Parrot to use
- Only `Application` section is required, the other sections are optional

Configuring Data Access

- Start with simple Read/Write storage access

```
[Directories]
export_base = /export/local
read = my_input_data
write = my_output_data

[Application]
location =
script = /usr/bin/hostname
arguments =
```

Location of directory Chirp exports

Application will have read access to this

Application will have read and write access to this

Important: directories given in both read and write settings are relative to directory in export_base

Specifying application execution

- Specifying which application to run and passing arguments to applications

```
[Directories]
export_base = /export/local
read = my_input_data
write = my_output_data

[Application]
location = http://my.domain/~user/myapp.tar.gz
script = ./myapp/run.sh
arguments =
```

URL to application tarball

Gives the command to run

Arguments to pass to script or binary, can also give arguments in job manager or grid submit file

Adding remote application access

- Slightly more advanced version with application access over CVMFS

CVMFS repository name

Options for repository access

```
[Directories]
export_base = /directory/to/share
write = my_input_data
read = my_output_data

[CVMFS]
repo1 = cvmfs.repo.name
repo1_options = url=http://repo.location/repo_path/,pubkey=http://
repo.key/location,proxies=squid_proxy:3128
repo1_key = http://repo.key/location

[Application]
location = http://my.host.name/~user/my_app.tar.gz
script = ./my_app/run_app.sh
arguments =
```

Location of CVMFS key

Using SkeletonKey and HT Condor

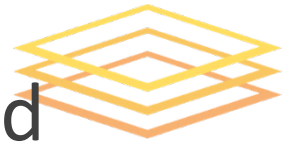
```
universe      = vanilla
executable    = ./job_script.sh
arguments     = $(Process)
notification = Error
input         =
output           = /tmp/my_job.out.$(Process)
error            = /tmp/my_job.err.$(Process)
log              = /tmp/my_job.log
should_transfer_files   = YES
when_to_transfer_output = ON_EXIT
queue 40
```

Shell script generated by SkeletonKey

Additional arguments passed to user script

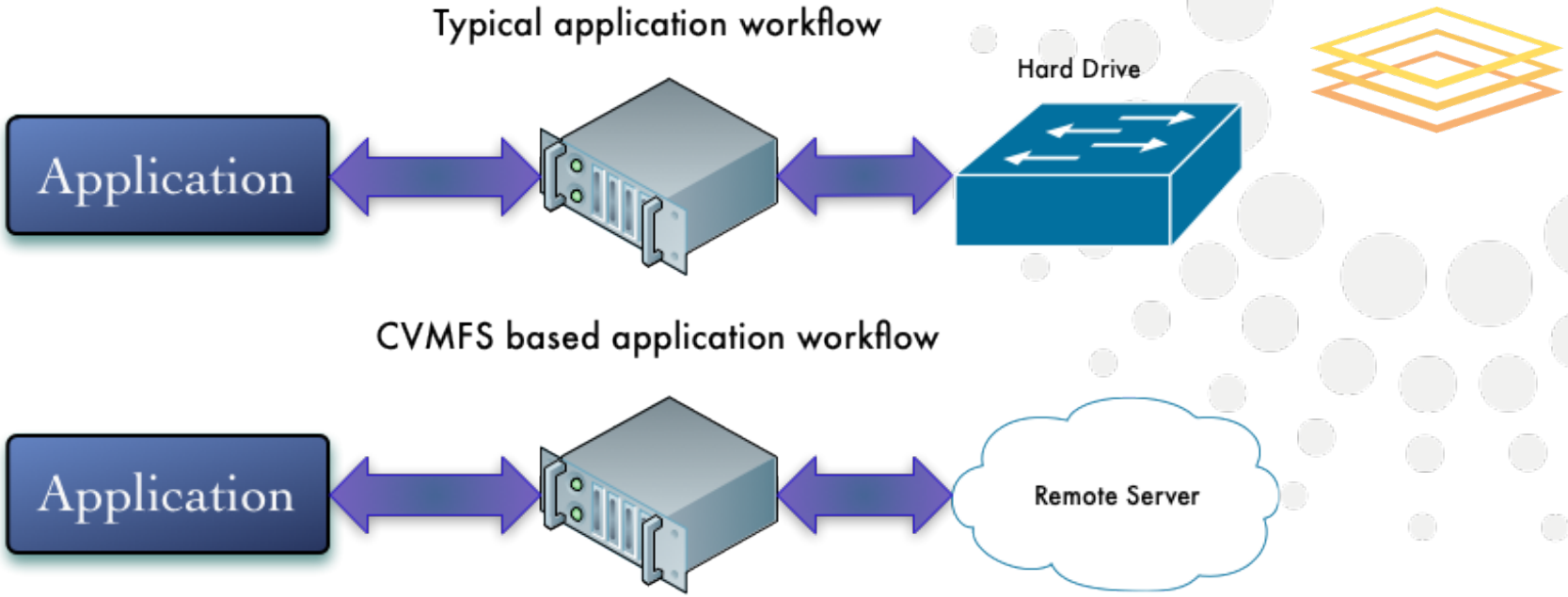
What's the performance?

- We were interested in determining what the overheads were to remote software access to see how viable this solution would be
  - Examined software runtimes when using different access methods
- Also interested in how Chirp/Parrot affected data access
  - Benchmarked streaming reads and writes to give information on this



Overhead for remote software access

- Compared the performance of R and Mathematica when running from local filesystem and from CVMFS



Application benchmark setup

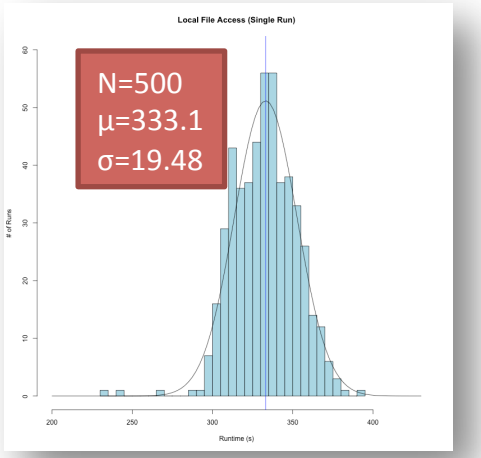
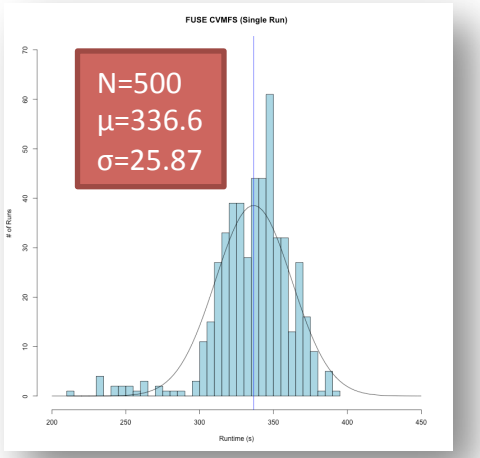
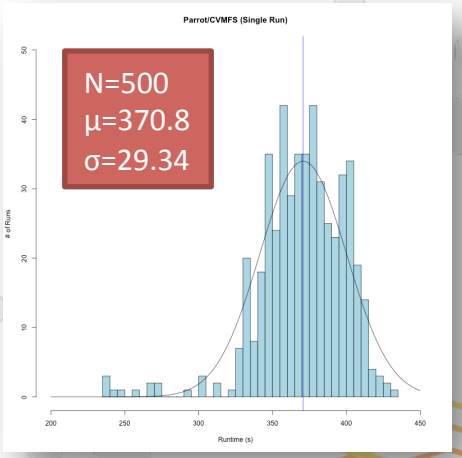
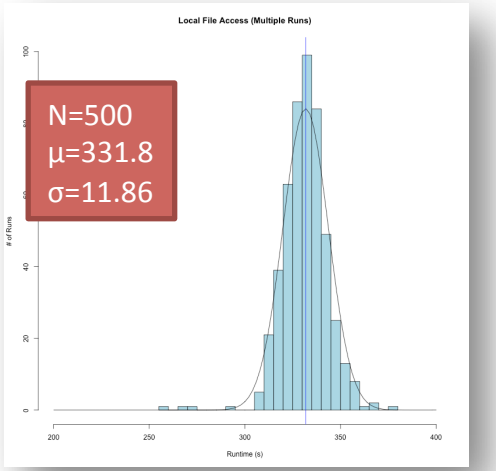
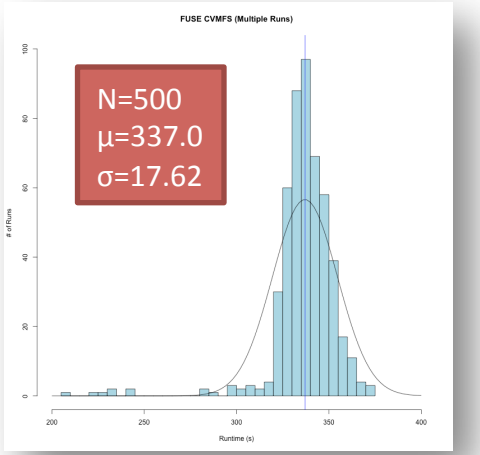
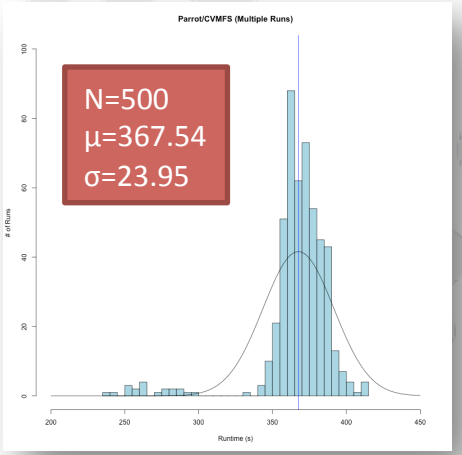
- CVMFS server: SL5.5 system running on a VM with 4GB of RAM and 4 virtual cores with a 1Gb network connection
- Applications run on the OSG-ITB cluster with 48 job slots on 3 compute nodes with 1Gb connections. Each slot has 2GB of RAM available.

Benchmarking R overheads

- R is commonly used in several fields to do statistical analysis.
  - We adapted example from a UC3 user
  - R script analyzes multi-layer geographical bitmap data

- Did two comparisons:
  - Single invocation of R to analyze 400 layers from a single file located in scratch space
  - Invoked R four times and analyzed 100 layers from a single file in scratch space during each invocation

Comparison of R runtimes

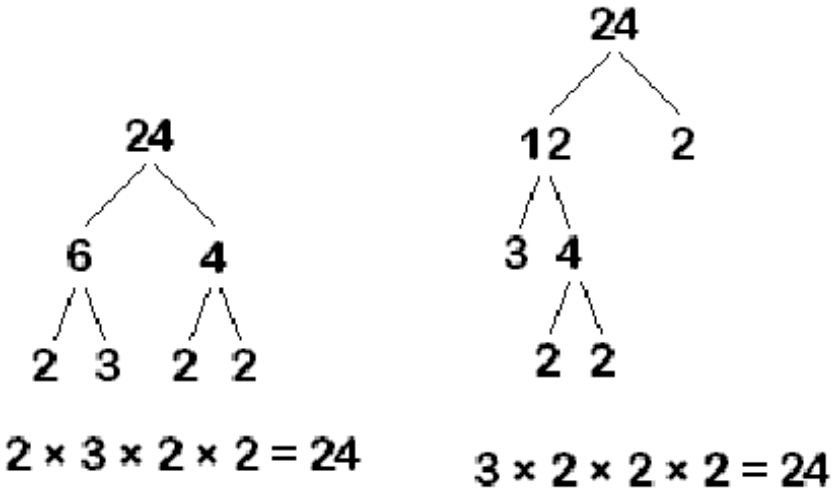
	Local Access	FUSE/CVMFS	Parrot/CVMFS
Single Invocation			
Multiple Invocations			

R Benchmarks Summarized

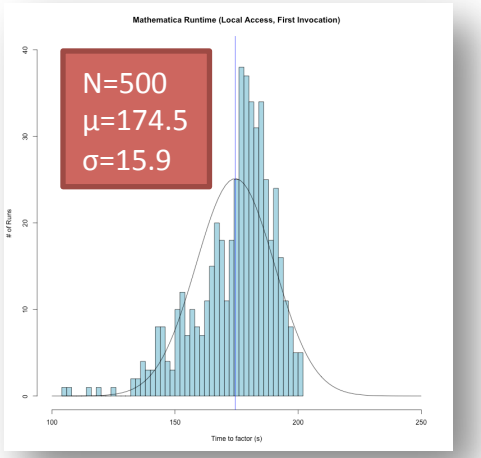
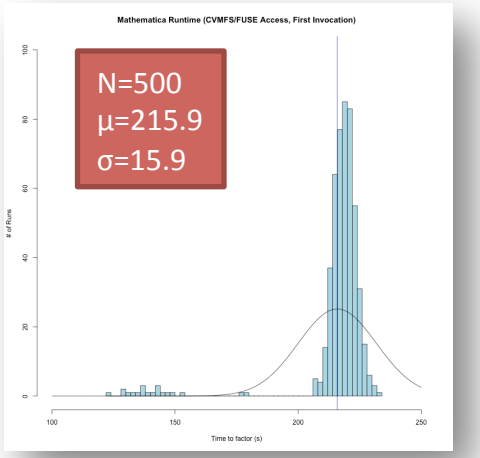
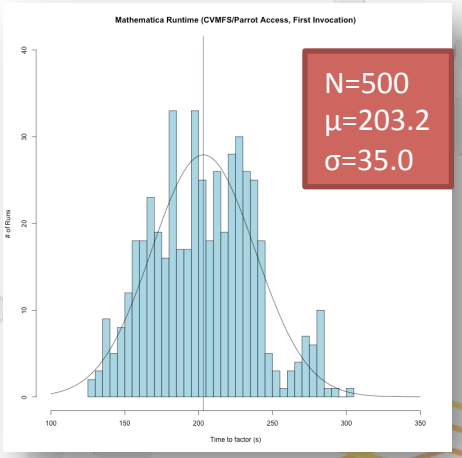
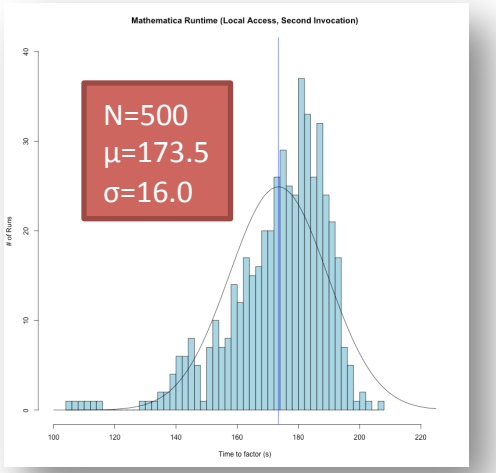
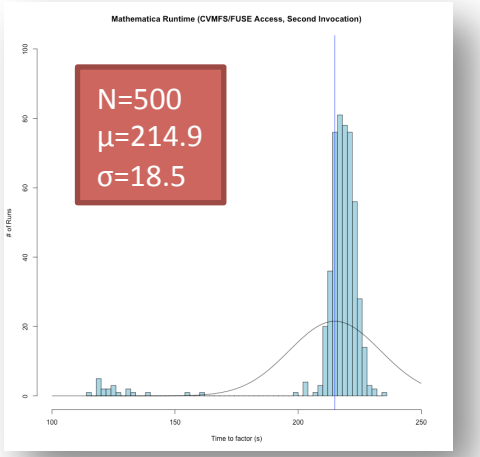
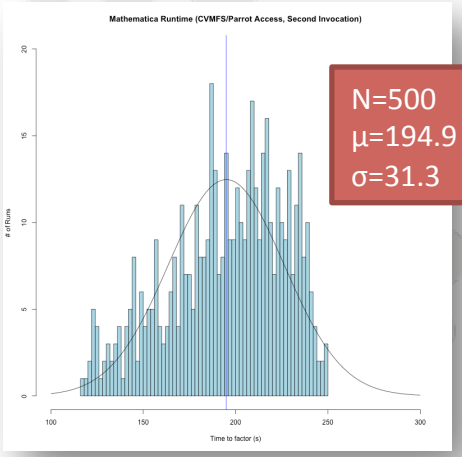
- Running R using Parrot/CVMFS takes 371±29s while running it on local filesystem takes about 333±19s
  - Overhead for using Parrot/CVMFS is only about 40s per invocation in this instance
- Running R multiple times to reduce latency (through cached copies) reduces overhead but not much (~7s) in this case
- Using FUSE/CVMFS gives a lower overhead but it's not always desirable to have to get each cluster's administrator to change their configuration when running in a distributed high throughput campus grid.

Do other applications have similar overheads?

- Also looked at Mathematica to see if this overhead was similar when using other applications
- Used a simple Mathematica script to time integer factorization of large numbers
- Ran factorization Mathematica twice to see how caching affects performance



Comparison of Mathematica runtimes

	Local Access	FUSE/CVMFS	Parrot/CVMFS
First Invocation			
Second Invocations			

Mathematica results summarized

- For this benchmark, Parrot/CVMFS performs fairly well
  - 15% slower than using Mathematica on local drive
  - Performance slightly better than FUSE/CVMFS but with much greater variation in performance
- Second run of Mathematica in the same session results in slightly better performance (~5%) but not significantly so

Overheads when accessing data remotely

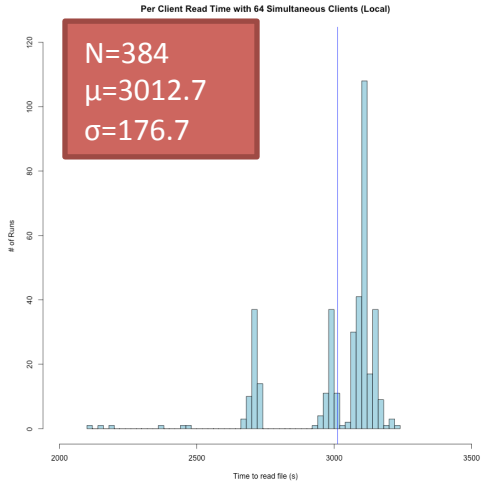
- Ran tests on two different clusters
- UC3 Chirp server:
  - Server with a 10Gb connection to switch ran a Chirp service to proxy access to three drive RAID0 array
- OSG-ITB Compute cluster:
  - Jobs run on OSG-ITB cluster to simulate remote client connections
  - Accessed UC3 storage using the Chirp server as a proxy
  - Nodes have 1Gb connection to switch

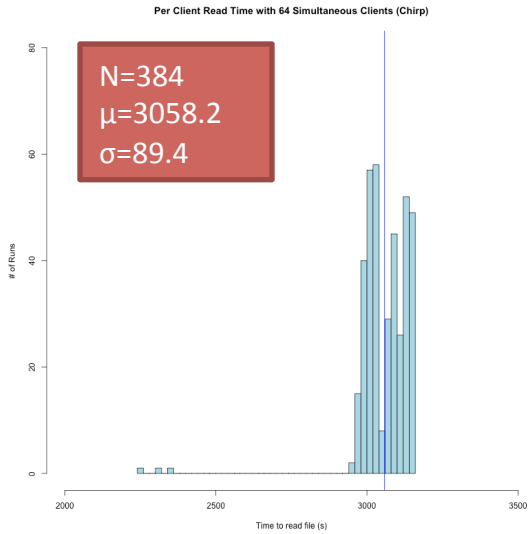
Quantifying remote data access overhead

- Tests designed to look at raw speed of storage subsystem used and then performance penalty imposed by access storage remotely
- Ran several tests to do this:
  - Bonnie++ tests head server to characterize performance of RAID0 array
  - Read/Write tests on Chirp server accessing RAID array directly
  - Read/Write tests on Chirp server with 64 applications accessing RAID array

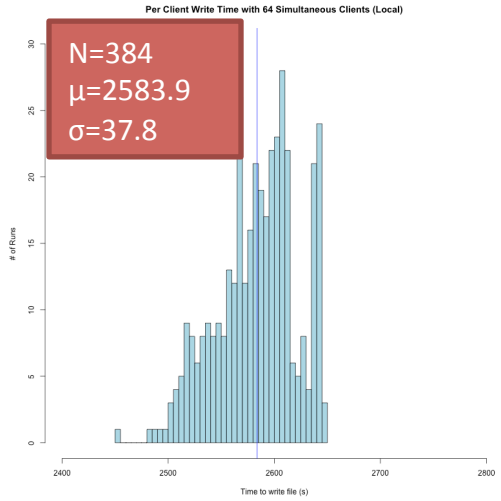
Performance using RAID Array

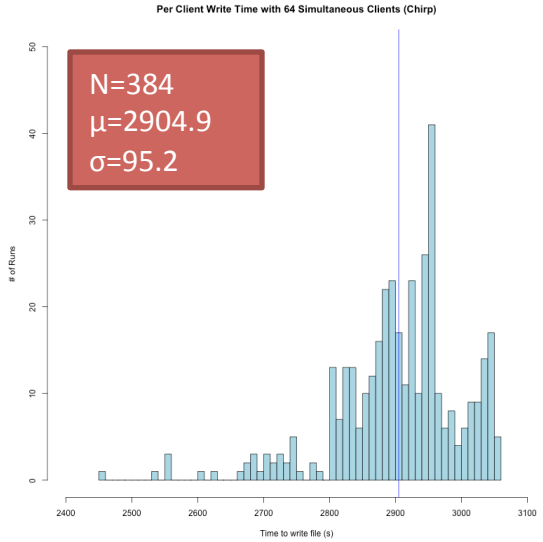
Read





Write





Local Access

Chirp

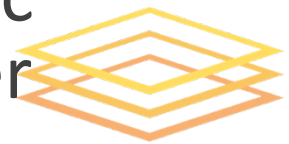
64 simultaneous clients reading or writing 10GB each to different files. Local access clients are running on Chirp server and accessing RAID array directly

Data Benchmark Conclusions

- Parrot/Chirp did not give an appreciable performance hit (~1%) when reading, performance difference a bit larger (~12%) when writing but not too large
  - Clients took an average of 3012s to read locally vs 3058s to read file using Parrot/Chirp.
  - Client took an average of 2584s to write locally vs 2904s to write using Parrot/Chirp
  - Using Parrot/Chirp compares very well to accessing array locally
- Aggregate bandwidth was about 180-230 MB/s on writes and 120-150MB/s on reads
- Bonnie++ tests on RAID array indicated streaming reads and writes got about 300-350MB/s
  - Chirp utilizes ~40-60% of max throughput to array

Conclusion

- SkeletonKey provides a convenient way to use Chirp and Parrot to remotely access data and software
- Performance using Parrot/Chirp for both data and software access within 10% or better on benchmarks tested
- Provides a viable method to gain opportunistic access to resources on campus grids and other distributed high throughput grids



- Future directions:
  - Expand to other users and add enhancements based on user feedback
- Questions?

Further information

- SkeletonKey:
  - Git: https://github.com/DHTC-Tools/UC3/tree/master/skeleton_key
  - Documentation: https://twiki.grid.iu.edu/bin/view/CampusGrids/SkeletonKey
- Chirp, Parrot, HDFS
  - Douglas Thain and Miron Livny, **Parrot: An Application Environment for Data-Intensive Computing,*Scalable Computing: Practice and Experience, 6(3), pages 9-18, September, 2005.***
  - Douglas Thain, Christopher Moretti, and Jeffrey Hemmes, **Chirp: A Practical Global Filesystem for Cluster and Grid Computing,*Journal of Grid Computing, 7(1), pages 51-72, March, 2009. DOI: 10.1007/s10723-008-9100-5***
  - Patrick Donnelly, Peter Bui, Douglas Thain, **Attaching Cloud Storage to a Campus Grid Using Parrot, Chirp, and Hadoop ,*IEEE International Conference on Cloud Computing Technology and Science, pages 488-495, November, 2010. DOI: 10.1109/CloudCom.2010.74***

Acknowledgements

- CCTools team, http://www.nd.edu/~ccl/
- Dan Bradley @ UW-Madison
- Colleagues at UC3:
  - Lincoln Bryant, Marco Mambelli, Rob Gardner