Building a complex user application for CMS with DAGMan

Brian Bockelman

Computing is Hard

- I will discuss personal experiences and lessons learned when building a new analysis system for CMS.
  - I selected **four favorite** problems I thought the community would find interesting.
  - I tell the problems, not the success storis.
- The project has a large team (including present and past members).
  - Each one would give a different point-of-view.
  - So consider my words an opinion, not the word of God.

What's in a CMS Analysis?

- For the purpose of presentation, a CMS analysis is:
  - **Input**: Some code tarball and configuration.
  - **Input**: Name of dataset.
  - **Output**: Dataset resulting from running the CMS application and user code on the input dataset.
- The rest is just details!

Ok, maybe more details

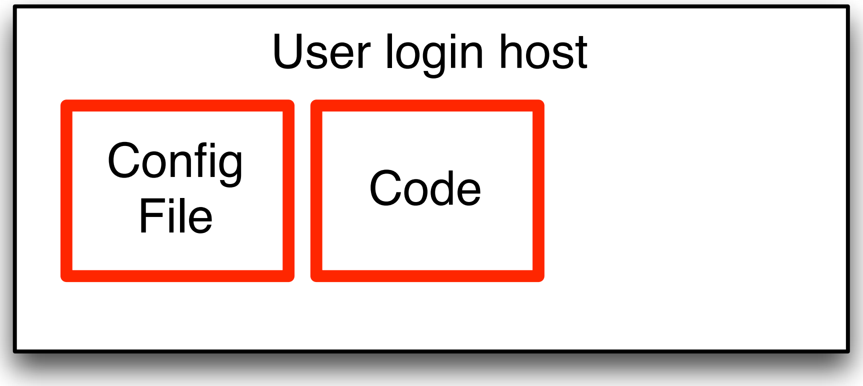
- A **dataset** is composed of one or more files.
- We break the analysis **task** into a set of **jobs**:
    - Each job can be run independently.
    - Each job reads in one or more files from the dataset.
        - The sites where the job can run are are determined from the location of the datasets.
    - Each job has a single result file that must return to a specified location.

Let’s Get CRABby

- CMS obviously has a system to do this - we found the Higgs, didn’t we?
- CRAB = CMS Remote Analysis Builder. Currently, CRAB2.
- The CRAB2 client runs on the user’s development node.
    - Splitting of task to jobs is done on client.
    - Job submits are done from client.
    - Job tracking (done? failed? needs resubmit?) done from client.
    - Results are copied from worker node to a user-specified remote storage system.
- CRAB2 allows multiple **job tracking backends**. >95% jobs use HTCondor, but about 10 different ones have been in existence.

CRAB2

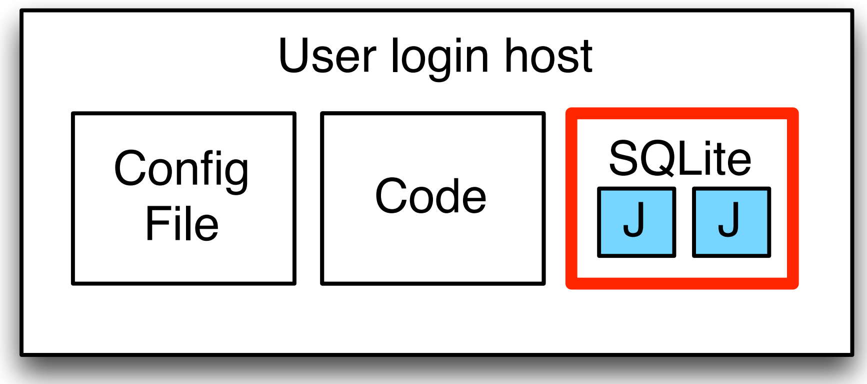
- User writes a **config file and code**.



CRAB2

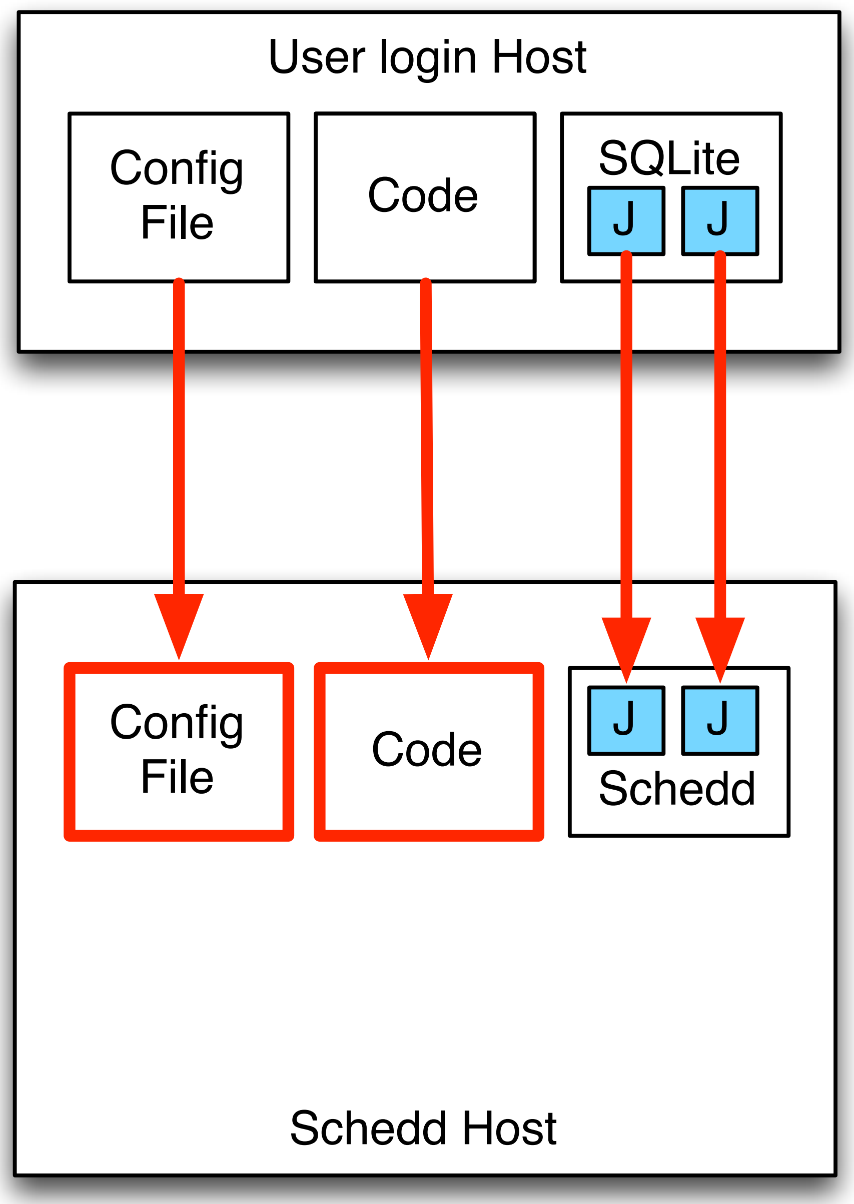
- User writes a config file and code.

- “**crab create**”



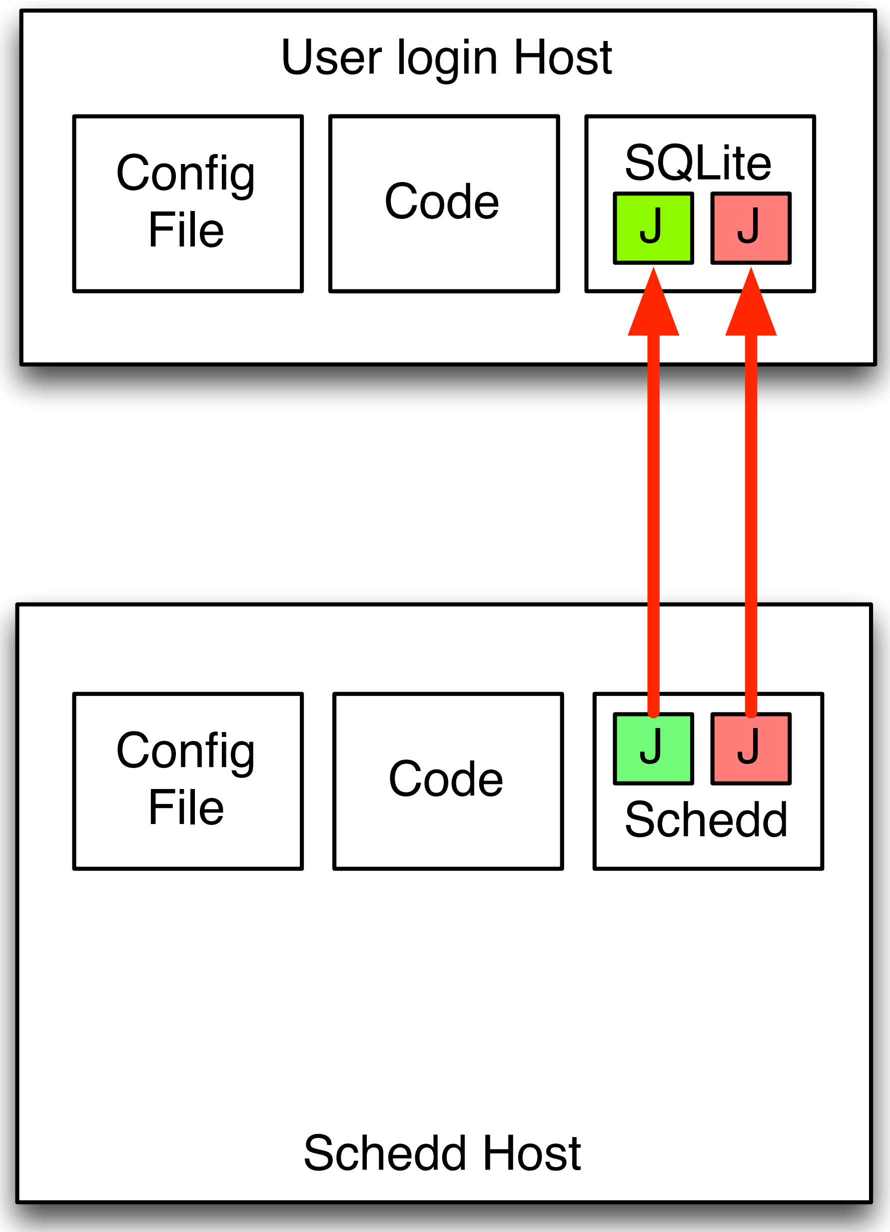
CRAB2

- User writes a config file and code.
- “crab create”
- “**crab submit**”



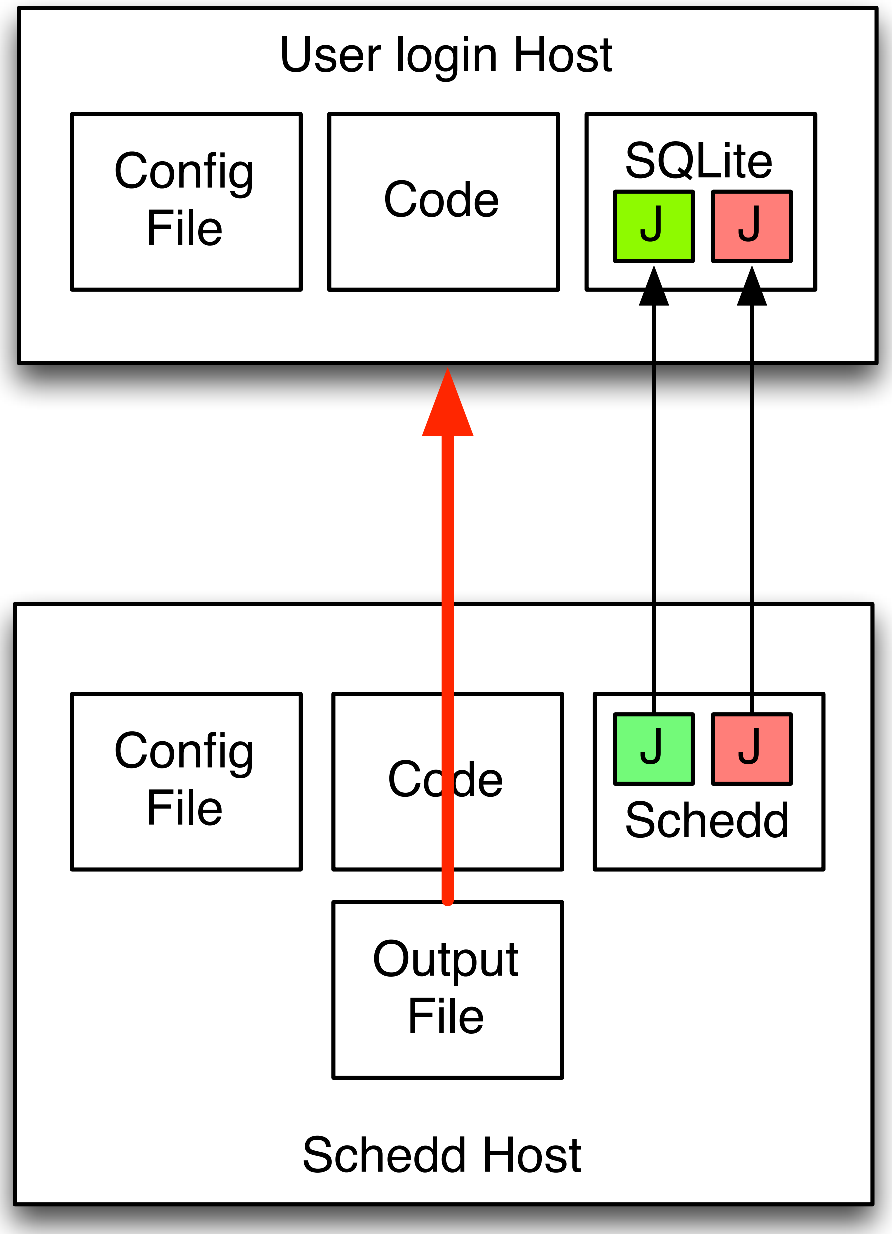
CRAB2

- User writes a config file and code.
- “crab create”
- “crab submit”
- “**crab status**” (times many)



CRAB2

- User writes a config file and code.
- “crab create”
- “crab submit”
- “crab status” (times many)
- “**crab getoutput**”
    - Retrieves stdout - job outputs go to storage service.



CRAB2 by the numbers

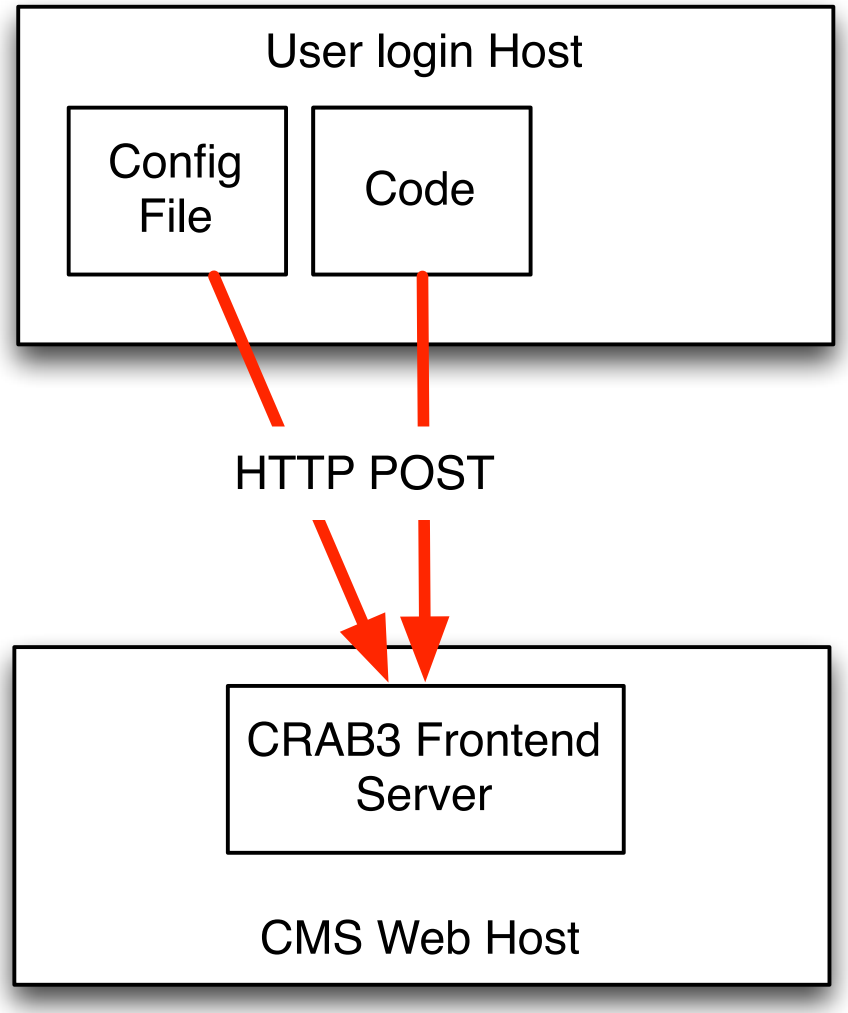
- Across the system, there are:
  - 200,000 jobs / day.
  - 25,000 cores used continuously.
  - 300 users.
  - 50 execution sites
  - One user support person.

What’s Wrong With This?

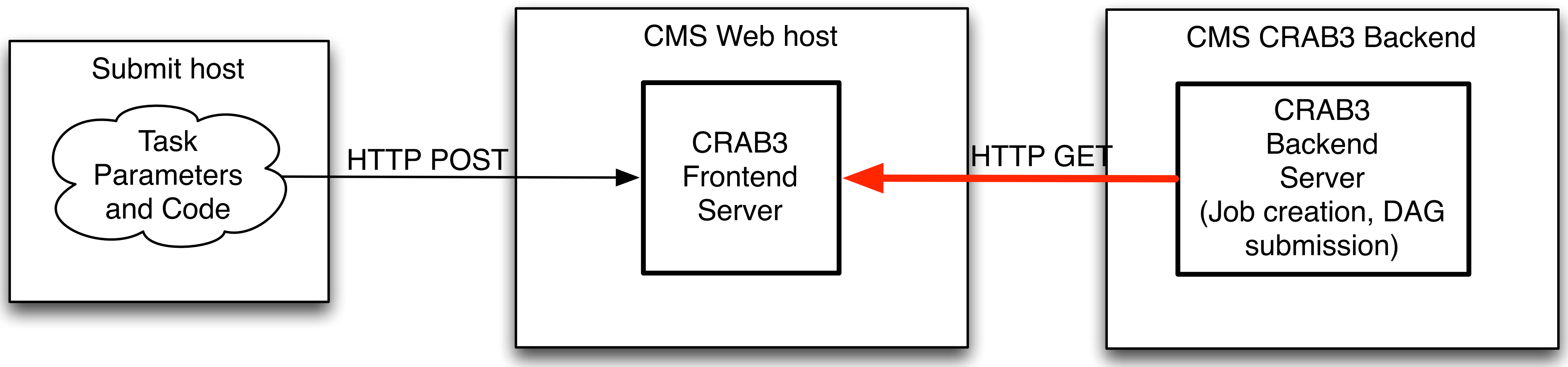
- CRAB2 has some seriously annoying flaws:
  - The job **“babysitter” is the user**, not a running process. [If a job fails 1 minute after you go to bed, it won’t be run until you wake up and press “resubmit”.]
  - **All logic is client-side**; users must install a new client to get bugfixes. Users *hate* installing updates.
  - **High rate of failures**, especially stageout. (More on this later)
- Each physics group has a hapless grad student whose role in life is to run that group’s workflows.

Enter CRAB3

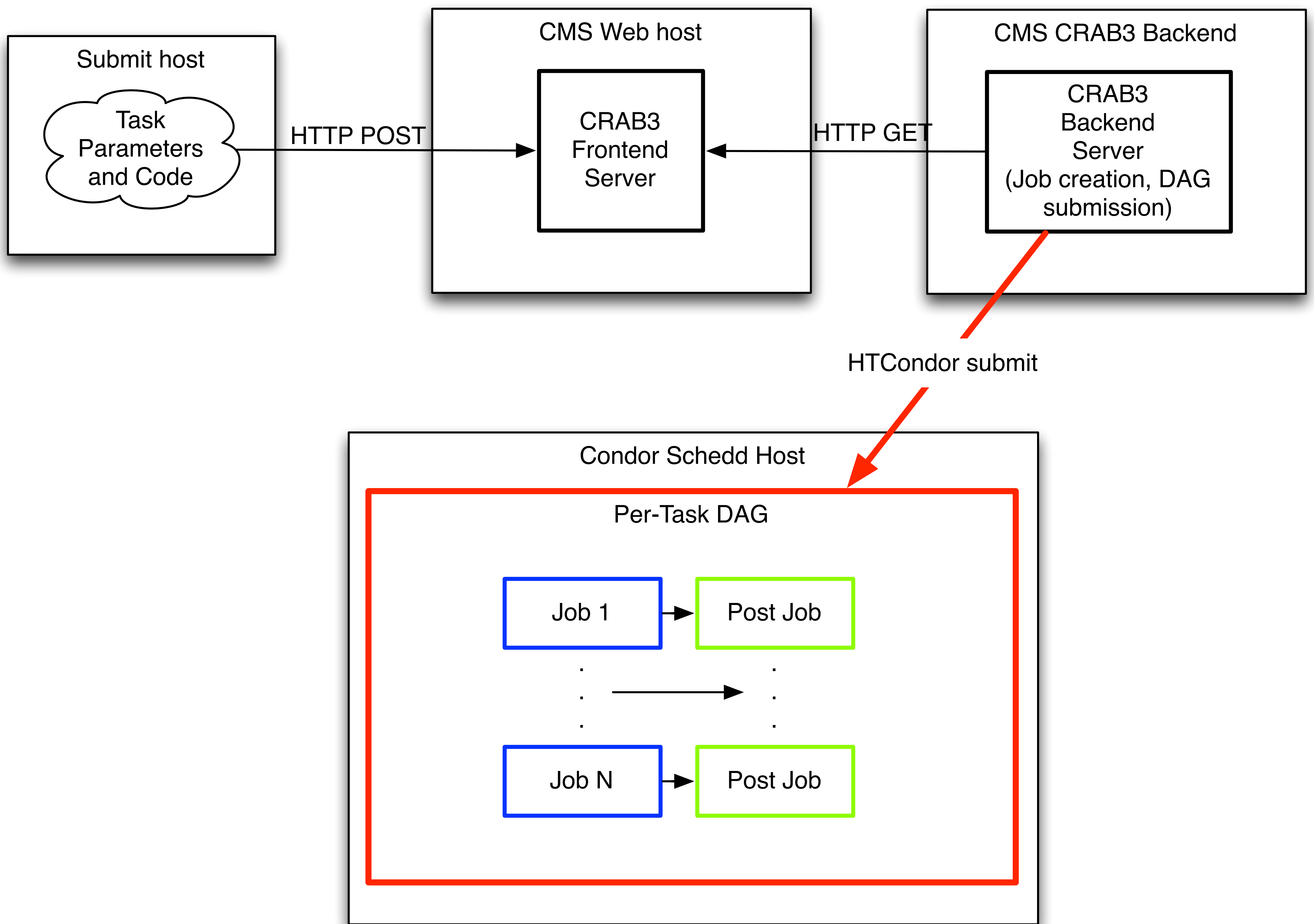
- The CRAB task is basically a DAG; CRAB2 relies on "DAGGradStudent".
- For CRAB3, we decided to have a backend which does *task tracking* instead of *job tracking*.
- We experimented with several task layers, then settled on DAGMan about 6 months ago.



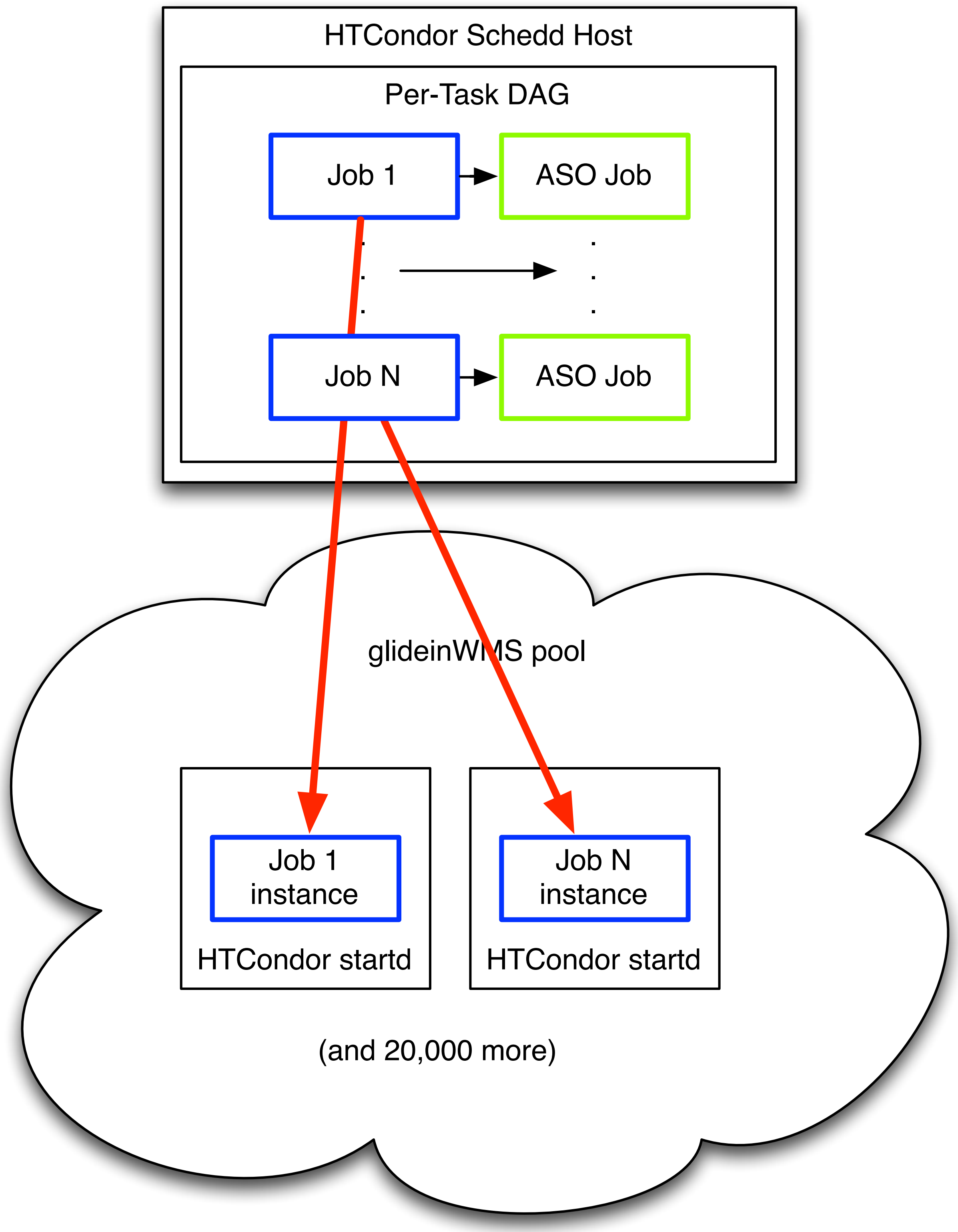
CRAB3 Architecture



CRAB3 Architecture



CRAB3 Architecture



What could possibly go wrong?

Interesting Problem 1: Remote Submit

- Strong push from CMS to centralize operations behind our central web portal.
- Submit hosts are operated by a separate team.
- Remote submit must somehow include user's X509 proxy certificate.

Solution: Python and MyProxy

- Python bindings provide the web portal with a mechanism to perform status queries. The backend server uses these for job submits.
- We have translated all the CRAB “verbs” to equivalent action on the DAGMan task in the schedd.
  - For example, if the DAG fails, the job goes onto HOLD; the equivalent of *condor_release* performs the resubmit.
- User uploads their X509 proxy to a separate MyProxy server and gives permission to the CRAB3 backend for retrieval; the backend then retrieves the proxy and pushes it to the schedd with the job submit.
  - We had to extend the python bindings to allow the backend to periodically push updated proxies to the schedd.

Interesting Problem 2: Task Interaction

- When running across 50 sites, failures are a fact of life. We use DAGMan post-scripts extensively to determine whether a failure should be retried.
- Inevitably, some jobs still fail (some errors require a human to examine or fix) permanently.
  - Consequently, the grad student then runs down the hall to yell at the sysadmin to fix things.
  - Once fixed, the grad student wants to be able to resubmit just that failed job!
  - Alternately, maybe the grad student would like to kill one DAG node while they investigate a job failure.
- Whether killing or resubmitting, we don’t want to upset the rest of the jobs! Hence, we don’t want to use rescue DAGs.

Solution

- **A wide series of hacks!**
- To kill individual running jobs, we simply remove them from the schedd queue.
- To kill the task, we hold the corresponding DAGMan job. This sends a SIGUSR1 signal to DAGMan, which removes all jobs from queue.
- To resubmit jobs, we change the hold signal to SIGKILL and hold/release the DAGMan job.
    - On release, the wrapper will rewrite the task's user log to change the exit code.
    - The condor_dagman process will restart and connect to running jobs using the log file.

Interesting Problem 3: Task Monitoring

- Each job in the user task is a node in a DAG. What's the state of the node?
    - If there is no corresponding job in the schedd, is it because the node failed? The node is successful? DAGMan has hit an idle job limit? Running a post-job?
    - Worse - *condor_q* is too heavyweight on the schedd to allow users to query directly for jobs in the schedd.
- How do we concisely present the relevant task information to users?
- How do we highlight potential problems?

Solution

- The DAGMan *node_status* file provides information about each node (one line per node in the DAG).
  - The file is placed in a web-accessible directory of the schedd; the CRAB frontend parses it and sends it to the user.
  - We worked closely with the HTCondor team to shake the bugs out of the handling of this file and deliver a new file format.
- We delivered a rewrite of the condor_q protocol to reduce the resources needed for queries.
- We wrote a custom monitoring application, glidemon, to summarize the task status.

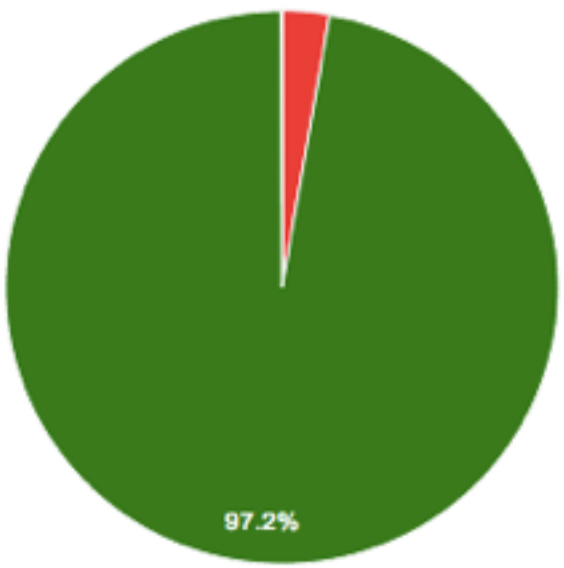
condor — root@submit-5:~ — ssh — 95×24

```
-bash-4.1$ crab status crab_xrootd_test_14 -u
Task name:                      140427_011655_vocms20:bbockelm_crab_xrootd_test_14
Task status:                    SUBMITTED
Monitoring URL:                 http://glidemon.web.cern.ch/glidemon/jobs.php?taskname=140427_0
11655_vocms20%3Abbockelm_crab_xrootd_test_14
Details:                        failed        37.8% (1430/3783)
                                finished      54.4% (2057/3783)
                                running        7.8% ( 296/3783)

Site Summary Table (including retries)

Site                     Runtime      Waste    Running Successful   Stageout     Failed
T1_DE_KIT              517:48:17 3210:55:46       13         97          0        253
T1_IT_CNAF              47:30:16  370:41:27        3         12          0         31
T1_TW_ASGC              28:18:49  104:23:49        1          6          0          8
T1_UK_RAL               62:13:12  782:06:28        1         15          0        111
T1_US_FNAL               0:00:00  275:14:05        0          0          0        393
T2_AT_Vienna            79:41:24  145:07:19        8          6          0         17
T2_BE_IIHE             600:19:45  426:31:39        5         63          0         32
T2_BE_UCL             1002:10:53 1450:34:27       69         63          0         97
T2_BR_SPRACE           632:12:39  385:16:44       13        114          0         27
T2_CH_CERN             135:03:54  571:53:50        3         23          0         70
T2_CH_CSCS             977:35:00 1683:48:52       40        172          0        152
T2_DE_DESY             324:51:02  839:32:14        6         49          0         93
```

Users

glidemon.web.cern.ch/glidemon/jobs.php?taskid=156165

Task 140425_194504_vocms20:bloom_crab_data_test_140425_private



(Only current jobs are shown. Show all jobs including resubmitted)

Summary of job failures

Top 5 Error Types

Number of jobs	Error Code (long)	Error Message
22	**65** (8001)	Other CMS Exception
2	**134** (134)	Abort (ANSI) or IOT trap (4.2 BSD)
1	**85** (8021)	FileReadError (May be a site error)

Top 5 Sites

Number of jobs	Site Name
23	T2_US_Purdue
1	T2_US_Nebraska
1	T2_UK_SGrid_RALPP

Individual job information (Log files)

Show 25 entries

Fast Filter:

Job Id	Id in Task	Version	Submit Server	Submit	Start	End	Site	Wall Time	Status
1516147.0	1	0	vocms20.cern.ch	25 Apr 21:46	25 Apr 22:01	25 Apr 22:02	T2_US_Purdue	00:01:41	**Completed** Logs: (Job, PostJob)
				25 Apr	26 Apr	26 Apr			**Completed**

Interesting Problem 4: Output file management

- Until now, I've only described half the battle!
- Depending on the user, outputs are somewhere between 1 and 3,000MB.
  - (Average job output size) X (# of jobs) = we can't use HTCondor file transfer.
  - Instead, the job output is copied to a storage system the user specifies.
- The #1 cause of failure in CRAB2 is **failed stageout**.

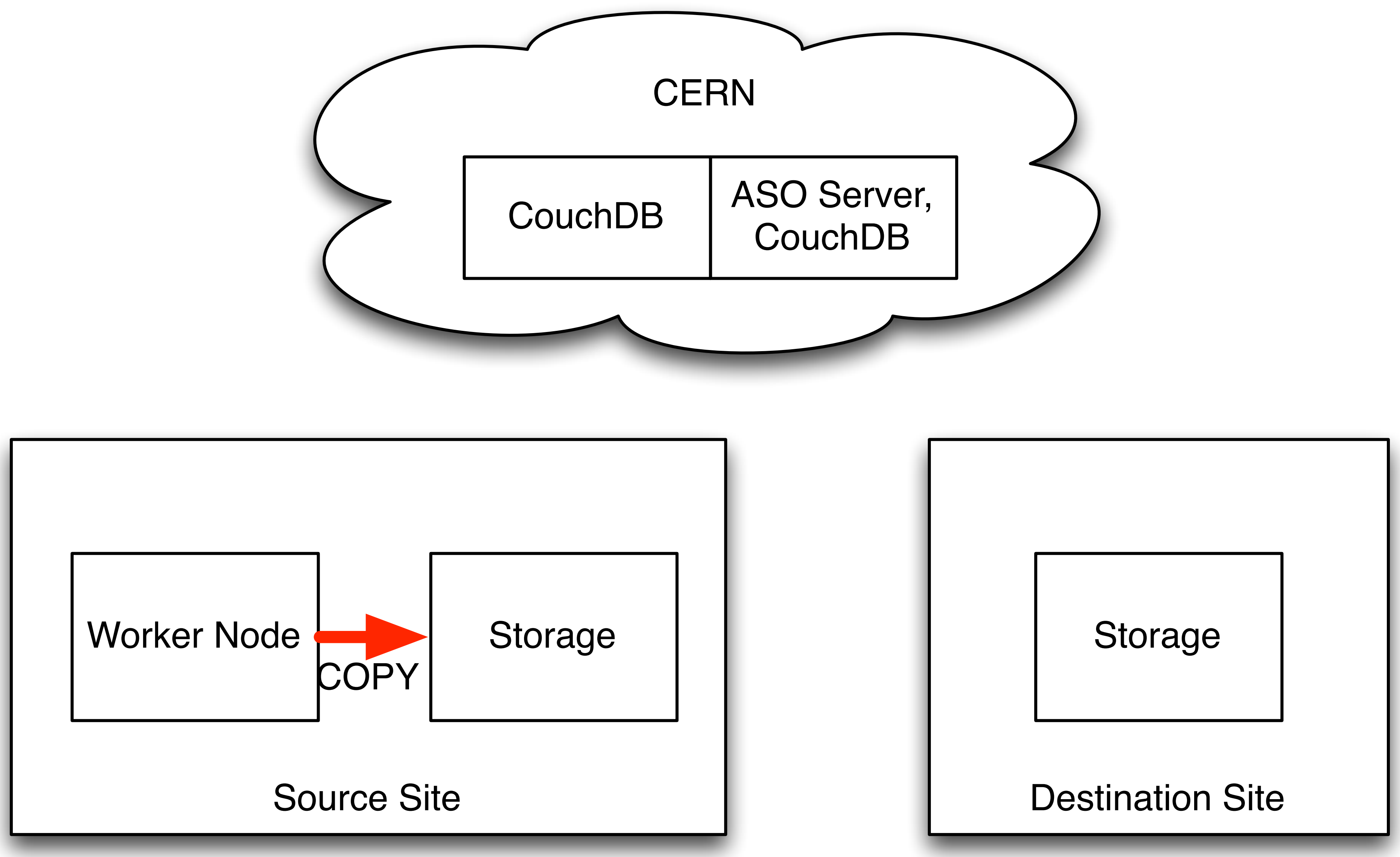
Introducing ASO

- To reduce error rates and improve CPU efficiency, the CRAB3 job wrapper copies output files to the “nearest” storage service, register the output file in a database, then exits.

- The Asynchronous StageOut (ASO) server will see the new output file and copy it to the user-specified storage service.

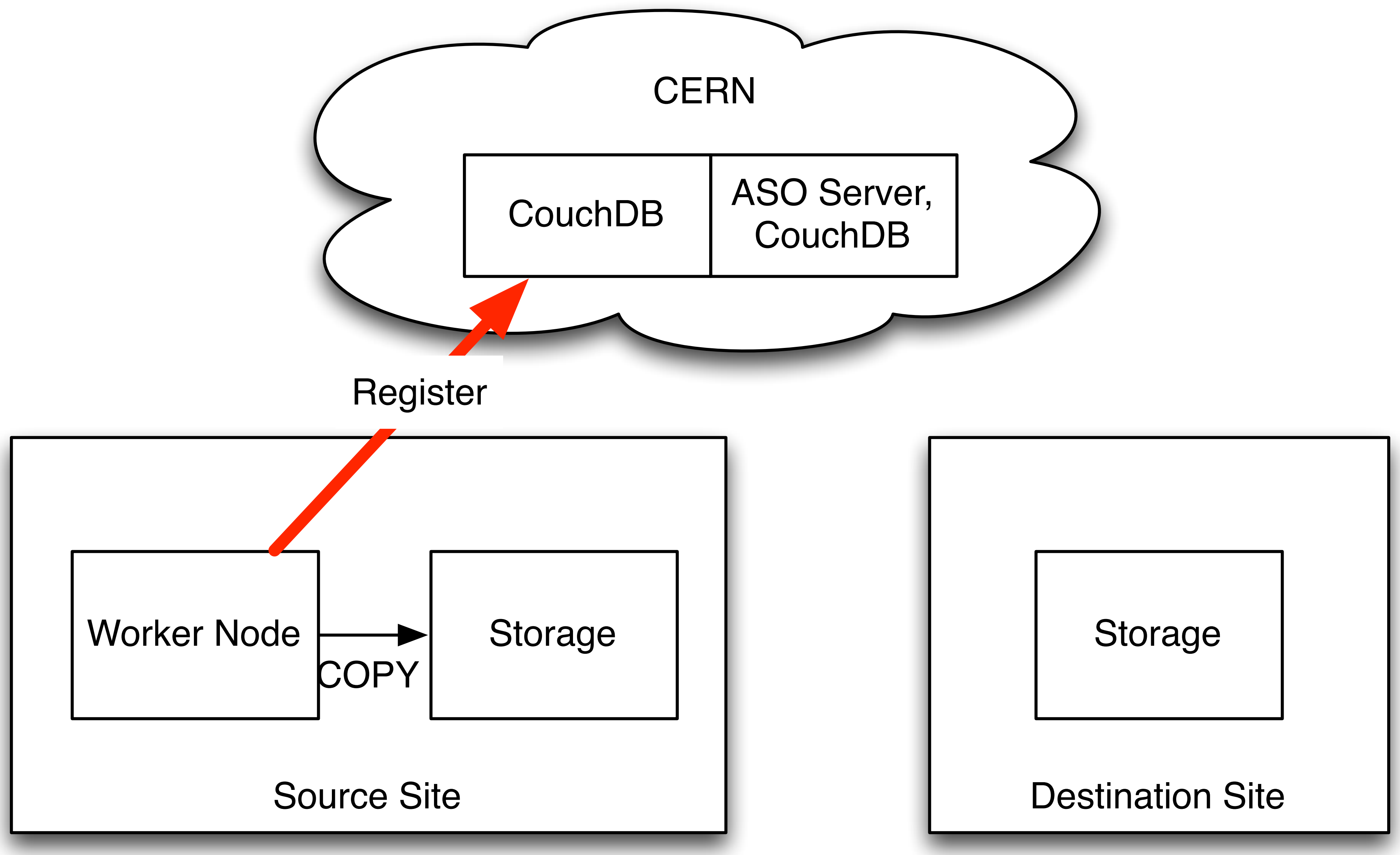
- Once the file is at the final location, the job can be marked as done.

ASO - Copy

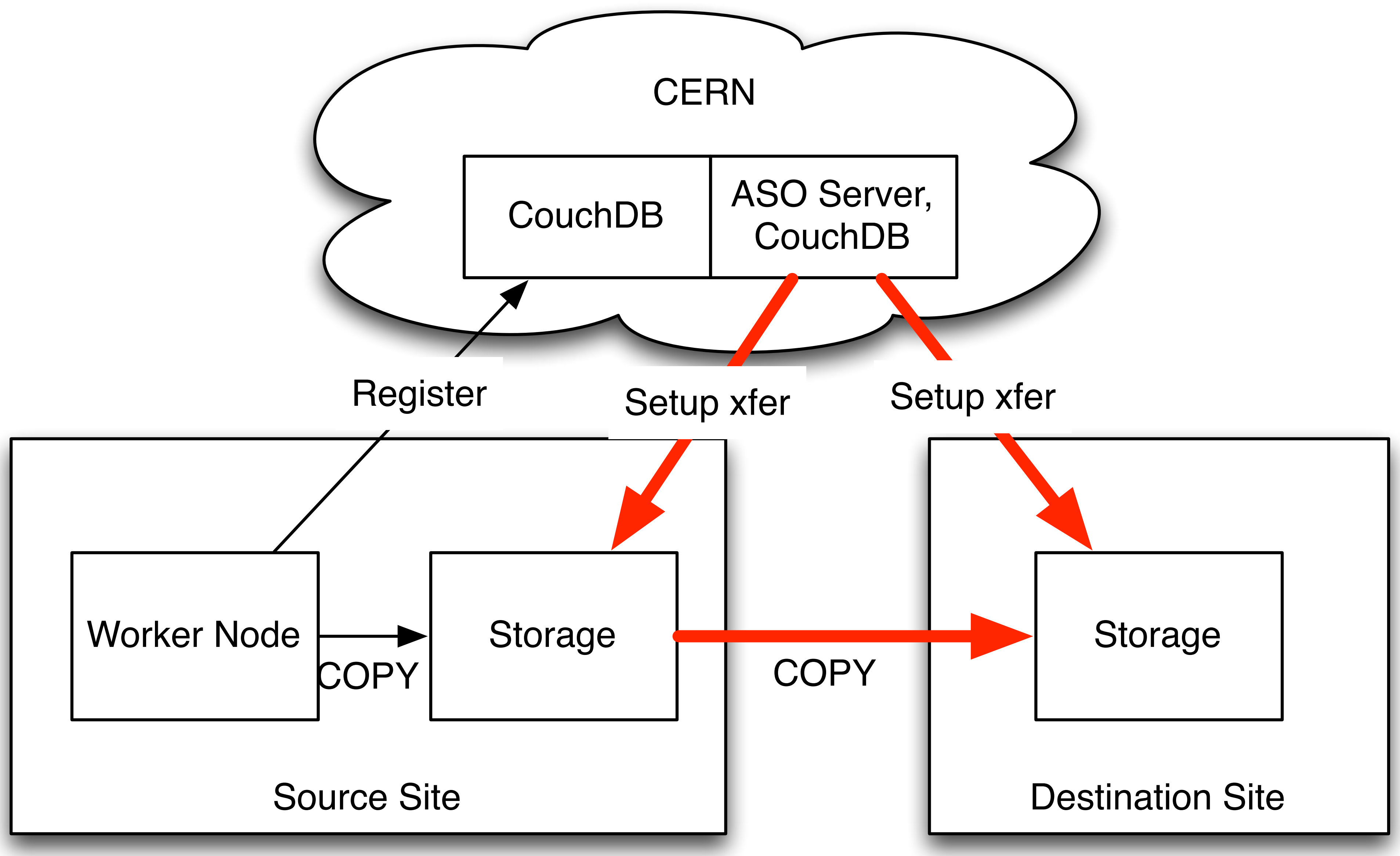


CERN
CouchDB
ASO Server, CouchDB
Worker Node
COPY
Storage
Source Site
Storage
Destination Site

ASO - Register



ASO - Copy to Remote



A Beautiful Can Of Worms

- **However** - DAGMan cannot mark a job as completed until ASO says the file has safely arrived.
  - Thus, the post-job must not exit until ASO says the transfer is done.
  - Each post-job takes up X MB of RAM, so we must limit the total number of post-jobs to Y per task.
  - Once Y post-jobs are running, *no other job can complete*.
  - Hence, the combination of DAGMan + ASO has the potential for causing head-of-line blocking behavior.
- We must have timeouts to make sure job completion progresses.

But wait, there's more!

- Essentially, we have two databases we must synchronize - ASO and DAGMan. All actions need to be idempotent or have a concurrency protocol.
- How do we prevent terabytes of data from accumulating at remote sites (flow control)?
  - CRAB2 had a crude and wasteful (but effective) way of throttling the production of output data - idle the CPU!
- The user and post-job need an indication of whether transfers are progressing.
  - Similar issue exists with running jobs.

Solutions

- We don’t have any here!
- We are looking forward to trying an upcoming experimental feature in 8.1.6 which allows the startd to start a new job while the current job is staging out.
    - Allows us to better throttle stageout without wasting CPU.
    - Reduces the number of services involved in the job lifetime - hope to reduce the surface area for bugs.

Conclusions

- CMS users don’t use DAGMan, they use CRAB3.
  - However, CRAB3 happens to rely on DAGMan for task management.
  - DAGMan automates the mundane tasks for us.
  - Ultimately, the biggest waste of resources was making physicists babysit jobs; wasted CPU was secondary.
- We’ve worked closely with the HTCondor team to iron out the biggest kinks.
  - While I focussed on the interesting problems, I left out quite a few mundane bug fixes the team delivered for CMS by the HTCondor team.
  - DAGMan and the python bindings have been hopefully made better for the entire community.
  - Quite a bit of work left to do!

Future Work

- There’s an immense amount of hackery around our kill/resubmit procedure; we would prefer to communicate directly with the DAGMan process.
    - We almost certainly don’t handle all the edge cases with killing the post-job.
- Our most pressing problem is the handling of the post-jobs.
    - The current best idea is to have a post-job return code that indicates DAGMan should re-run the post-job at a later time.
- Users hate “stuck” jobs; we want to do a better job of providing feedback by allowing *condor_tail* or updating the job ad with the # of events processed.