

---

# Grid Compute Resources and Job Management



**Open Science Grid**

---

# How do we access the grid ?

- ❑ Command line with tools that you'll use
- ❑ Specialised applications
  - Ex: Write a program to process images that sends data to run on the grid as an inbuilt feature.
- ❑ Web portals
  - I2U2
  - SIDGrid

# Grid Middleware glues the grid together

- *A short, intuitive definition:*

the software that glues together different clusters into a grid, taking into consideration the socio-political side of things (such as common policies on who can use what, how much, and what for)

# Grid middleware

- Offers services that couple **users** with **remote resources** through **resource brokers**
- Remote process management
- Co-allocation of resources
- Storage access
- Information
- Security
- QoS

# Globus Toolkit

- the *de facto* standard for grid middleware.
- Developed at ANL & UChicago (Globus Alliance)
- Open source
- Adopted by different scientific communities and industries
- Conceived as an open set of architectures, services and software libraries that support grids and grid applications
- Provides services in major areas of distributed systems:
  - ❑ Core services
  - ❑ Data management
  - ❑ Security

# Globus - core services

- Are the basic infra-structure needed to create grid services
- Authorization
- Message level security
- System level services (e.g., monitoring)
- Associated data management provides file services
  - GridFTP
  - RFT (Reliable File Transfer)
  - RLS (Replica Location Service)
- Globus uses GT4
  - Promotes open high-performance computing (HPC)

# Local Resource Managers (LRM)

- Compute resources have a **local resource manager** (LRM) that controls:
  - Who is allowed to run jobs
  - How jobs run on a specific resource
  - Specifies the order and location of jobs
- *Example policy:*
  - Each cluster node can run one job.
  - If there are more jobs, then they must wait in a queue
- LRMs allow nodes in a cluster can be **reserved** for a specific person
- *Examples:* PBS, LSF, Condor

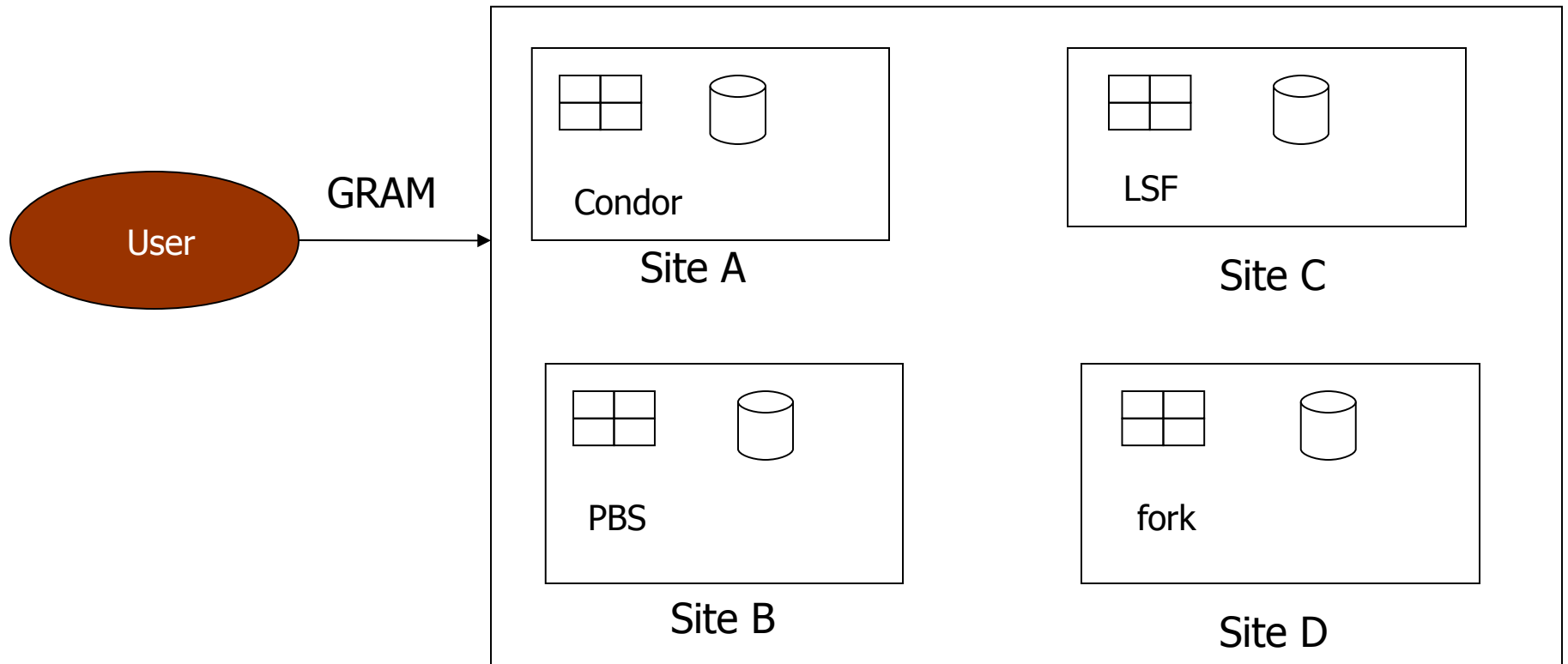
# GRAM

## Globus Resource Allocation Manager

- **GRAM** = provides a standardised interface to submit jobs to LRMs.
- Clients submit a job request to GRAM
- GRAM translates into something a(ny) LRM can understand
  - .... Same job request can be used for many different kinds of LRM



# Job Management on a Grid



The Grid

# Two versions of GRAM

- There are two versions of GRAM
  - GT2
    - Own protocols
    - Older
    - More widely used
    - No longer actively developed
  - GT4
    - Web services
    - Newer
    - New features go into GRAM4
  
- In this module, will be using GT2

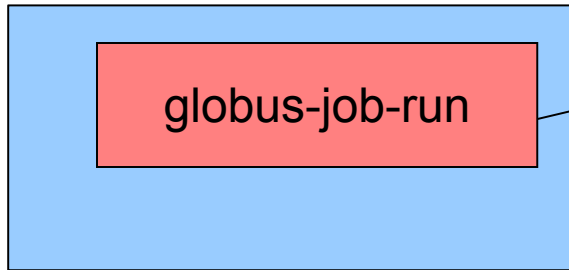
# GRAM's abilities

- Given a job specification:
  - ❑ Creates an environment for the job
  - ❑ Stages files to and from the environment
  - ❑ Submits a job to a local resource manager
  - ❑ Monitors a job
  - ❑ Sends notifications of the job state change
  - ❑ Streams a job's stdout/err during execution

# GRAM components

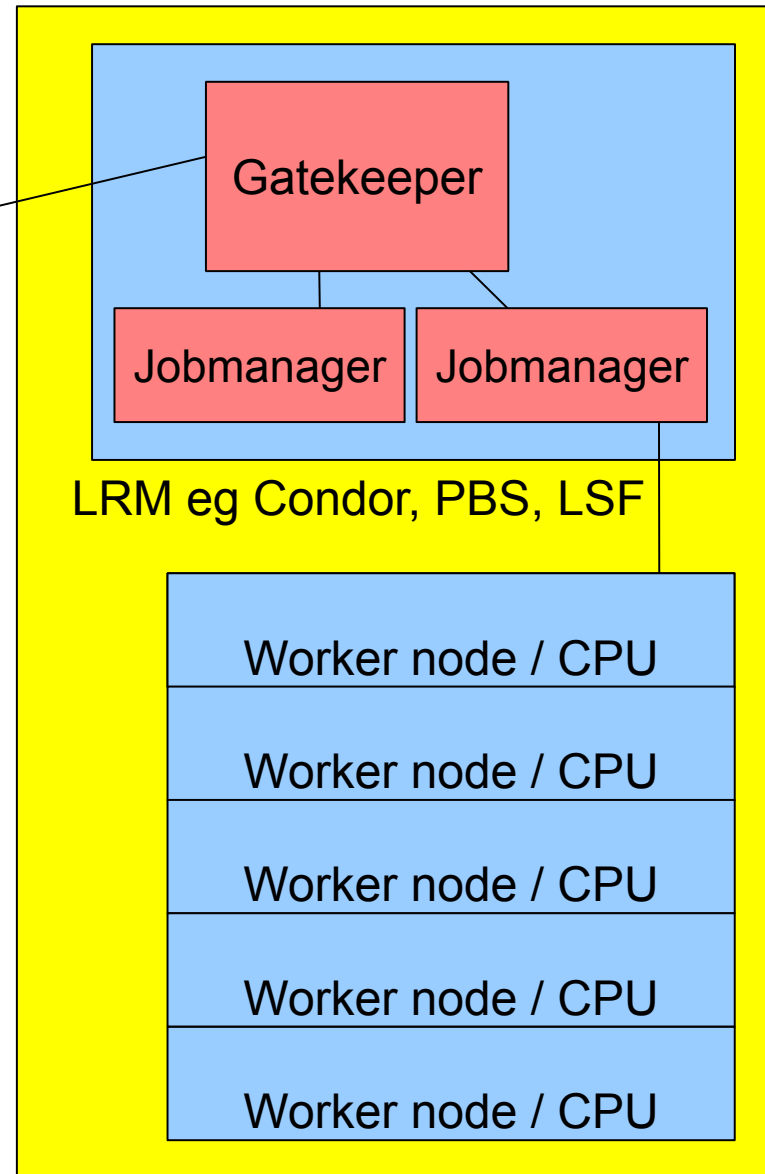
- Clients –
  - eg. globus-job-submit, globus-run
- Gatekeeper
  - Server
  - Accepts job submissions
  - Handles security
- Jobmanager
  - Knows how to send a job into the local resource manager
  - Different job managers for different LRMs

# GRAM components

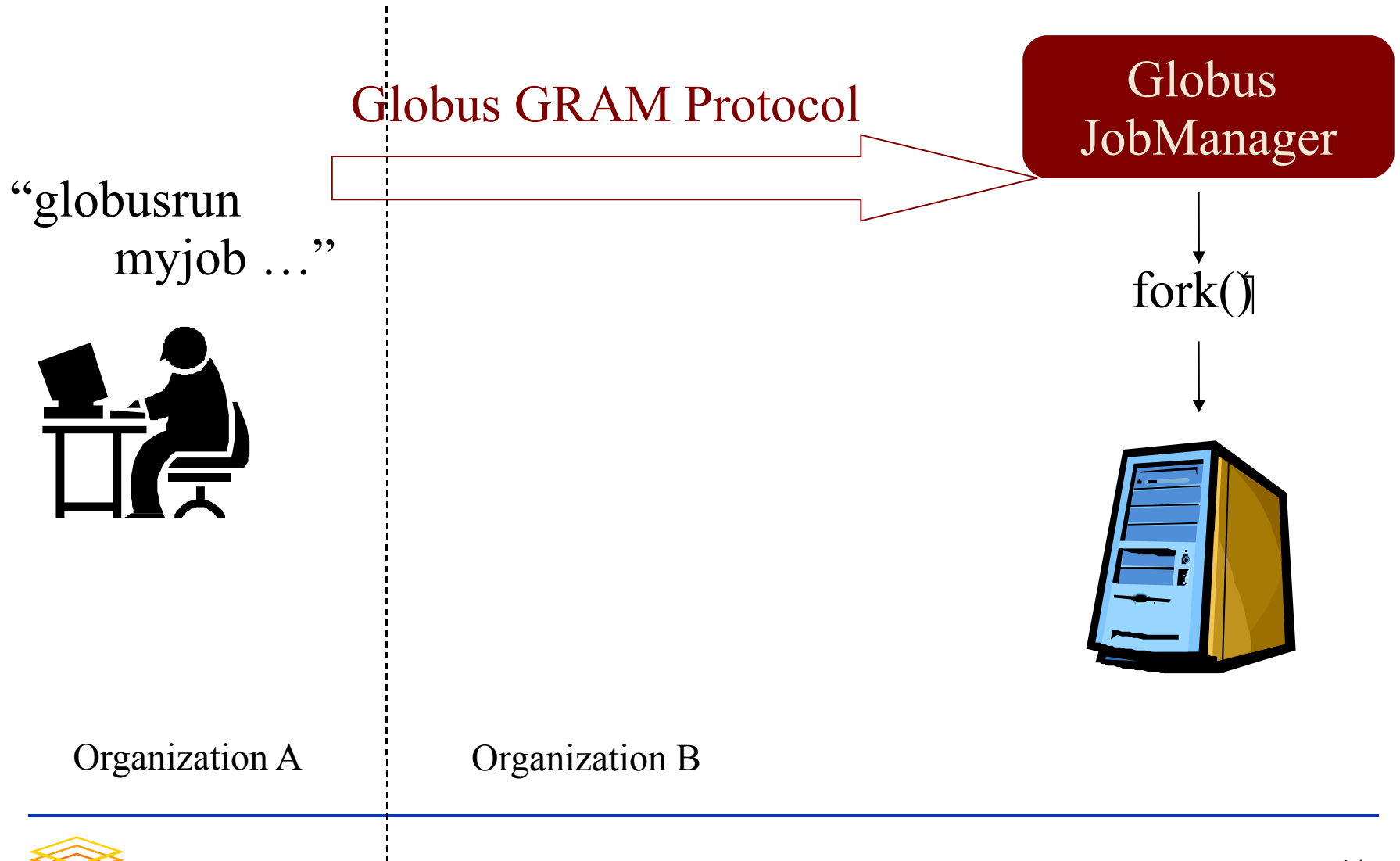


Submitting machine  
(e.g. User's workstation)

Internet



# Remote Resource Access: Globus



# Submitting a job with GRAM

- **globus-job-run** command

```
$ globus-job-run rookery.uchicago.edu /bin/hostname
```

- Run '/bin/hostname' on the resource rookery.uchicago.edu

- We don't care what LRM is used on 'rookery'. This command works with any LRM.

# The client can describe the job with GRAM's Resource Specification Language (RSL)

- Example:

```
&(executable = a.out)
(directory = /home/nobody )
(arguments = arg1 "arg 2")
```

- Submit with:

```
globusrun -f spec.rsl -r
rookery.uchicago.edu
```



# Use other programs to generate RSL

- RSL job descriptions can become very complicated
- We can use other programs to generate RSL for us
  - Example: Condor-G – next section

# Condor

- Condor is a specialized workload management system for compute-intensive jobs.
- is a software system that creates an HTC environment
  - Created at [UW-Madison](#)
  - Detects machine availability
  - Harnesses available resources
  - Uses remote system calls to send R/W operations over the network
  - Provides powerful resource management by *matching* resource owners with consumers (broker)

# How Condor works

## Condor provides:

- a job queueing mechanism
- scheduling policy
- priority scheme
- resource monitoring, and
- resource management.

Users **submit** their serial or parallel jobs to Condor,

Condor places them into a **queue**,

... chooses **when** and **where** to run the jobs based upon a policy,

... carefully **monitors** their progress, and

... ultimately **informs** the user upon completion.

# Condor - features

- Checkpoint & migration
- Remote system calls
  - Able to transfer data files and executables across machines
- Job ordering
- *Job requirements and preferences can be specified via powerful expressions*

# Condor lets you manage a large number of jobs.

- Specify the jobs in a file and submit them to Condor
- Condor runs them and keeps you notified on their progress
  - Mechanisms to help you manage huge numbers of jobs (1000's), all the data, etc.
  - Handles inter-job dependencies (DAGMan)
- Users can set Condor's job priorities
- Condor administrators can set user priorities
- Can do this as:
  - Local resource manager (LRM) on a compute resource
  - Grid client submitting to GRAM (as Condor-G)

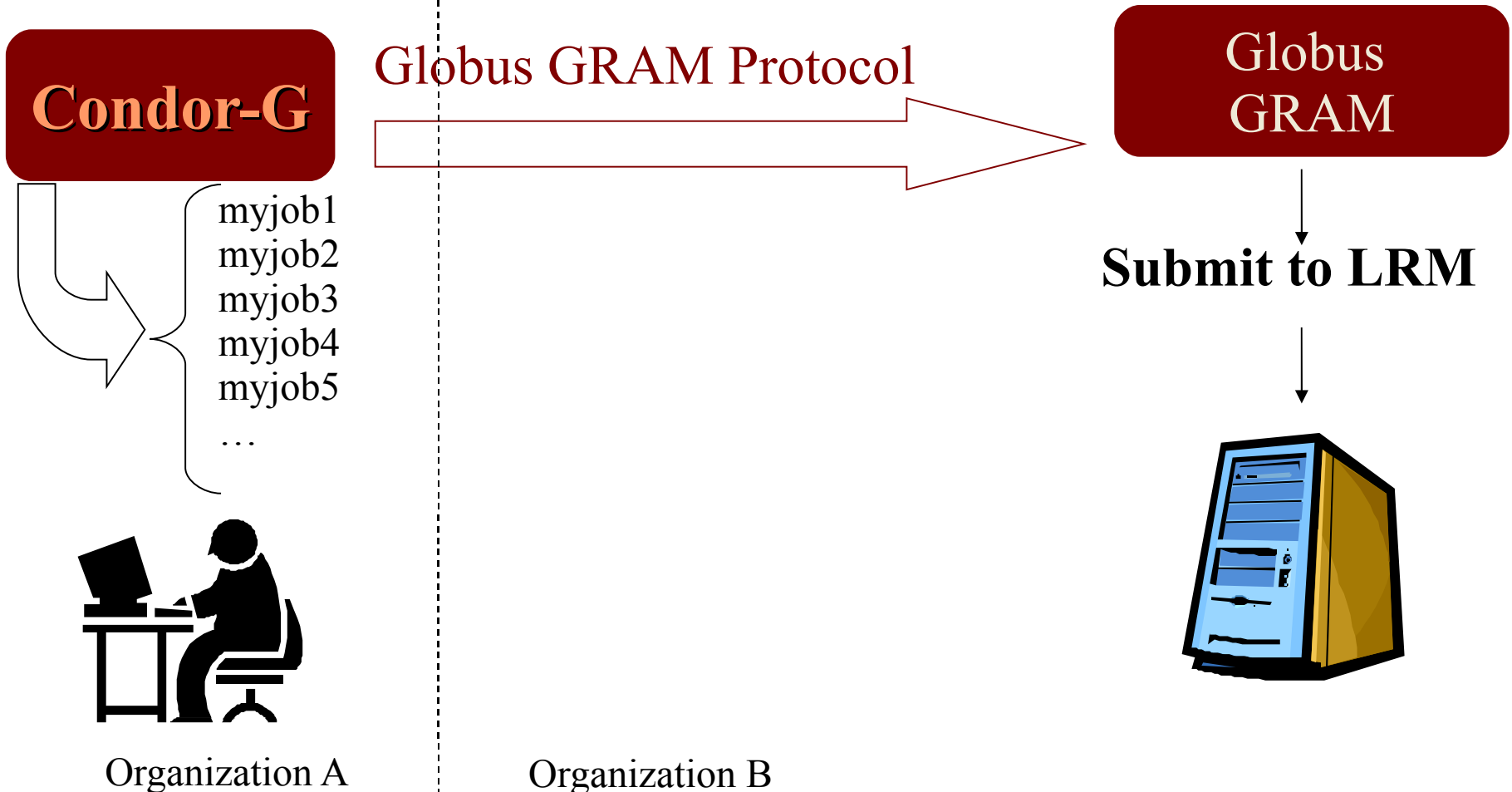
# Condor-G

- is the job management part of Condor.
- *Hint:* Install Condor-G to submit to resources accessible through a Globus interface.
- Condor-G does not *create* a grid service.
- It only deals with *using* remote grid services.

# Condor-G ...

- does whatever it takes to run your jobs, even if ...
  - ❑ The gatekeeper is temporarily unavailable
  - ❑ The job manager crashes
  - ❑ Your local machine crashes
  - ❑ The network goes down

# Remote Resource Access: Condor-G + Globus + Condor





# Condor-G: Access non-Condor Grid resources



- middleware deployed across entire Grid
- remote access to computational resources
- dependable, robust data transfer
- job scheduling across multiple resources
- strong fault tolerance with checkpointing and migration
- layered over Globus as “personal batch system” for the Grid

# Four Steps to Run a Job with Condor

- These choices tell Condor
  - ❑ **how**
  - ❑ **when**
  - ❑ **where** to run the job,
  - ❑ and describe exactly **what** you want to run.
- Choose a Universe for your job
- Make your job batch-ready
- Create a *submit description* file
- Run *condor\_submit*

# I. Choose a Universe

- There are many choices
  - **Vanilla:** any old job
  - **Grid:** run jobs on the grid
  - **Standard:** checkpointing & remote I/O
  - **Java:** better for Java jobs
  - **MPI:** Run parallel MPI jobs
  - Virtual Machine: Run a virtual machine as job
  - ...
- *For now we'll just consider vanilla*

## 2. Make your job batch-ready

- Must be able to run in the background:
  - no interactive input, windows, GUI, etc.
- Condor is designed to run jobs as a batch system, with pre-defined inputs for jobs
- Can still use `STDIN`, `STDOUT`, and `STDERR` (the keyboard and the screen), but files are used for these instead of the actual devices
- Organize data files

# 3. Create a Submit Description File

- A plain ASCII text file
- Condor does not care about file extensions
- Tells Condor about your job:
  - Which executable to run and where to find it
  - Which universe
  - Location of input, output and error files
  - Command-line arguments, if any
  - Environment variables
  - Any special requirements or preferences

# Simple Submit Description File

```
# myjob.submit file
# Simple condor_submit input file
# (Lines beginning with # are comments)
# NOTE: the words on the left side are not
#       case sensitive, but filenames are!
Universe      = vanilla
Executable    = analysis
Log           = my_job.log
Queue
```

## 4. Run `condor_submit`

- You give *condor\_submit* the name of the submit file you have created:

```
condor_submit my_job.submit
```

- *condor\_submit* parses the submit file

# Another Submit Description File

```
# Example condor_submit input file
# (Lines beginning with # are comments) ↵
# NOTE: the words on the left side are not
#       case sensitive, but filenames are!
Universe      = vanilla
Executable    = /home/wright/condor/my_job.condor
Input         = my_job.stdin
Output        = my_job.stdout
Error         = my_job.stderr
Arguments     = -arg1 -arg2
InitialDir    = /home/wright/condor/run_1
Queue
```



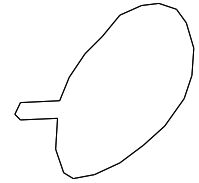
# Details

- Lots of options available in the submit file
- Commands to
  - watch the queue,
  - the state of your pool,
  - and lots more
- You'll see much of this in the hands-on exercises.

# Other Condor commands

- `condor_q` – show status of job queue
- `condor_status` – show status of compute nodes
- `condor_rm` – remove a job
- `condor_hold` – hold a job temporarily
- `condor_release` – release a job from hold

# Submitting more complex jobs



- express dependencies between jobs  
⇒ WORKFLOWS
- And also, we would like the workflow to be managed even in the face of failures

# Want other Scheduling possibilities?

## Use the Scheduler Universe

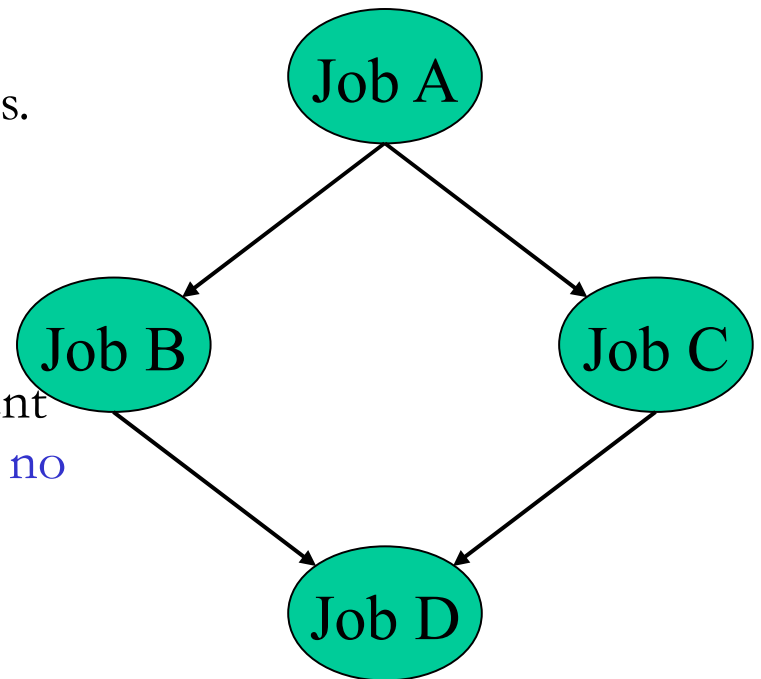
- In addition to VANILLA, another job universe is the *Scheduler Universe*.
- Scheduler Universe jobs run on the submitting machine and serve as a meta-scheduler.
- **Condor's Scheduler Universe lets you set up and manage job workflows.**
- DAGMan meta-scheduler included
  - DAGMan manages these jobs

# DAGMan

- **Directed Acyclic Graph Manager**
- DAGMan allows you to specify the *dependencies* between your Condor jobs, so it can *manage* them automatically for you.
- (e.g., “Don’t run job “B” until job “A” has completed successfully.”)

# What is a DAG?

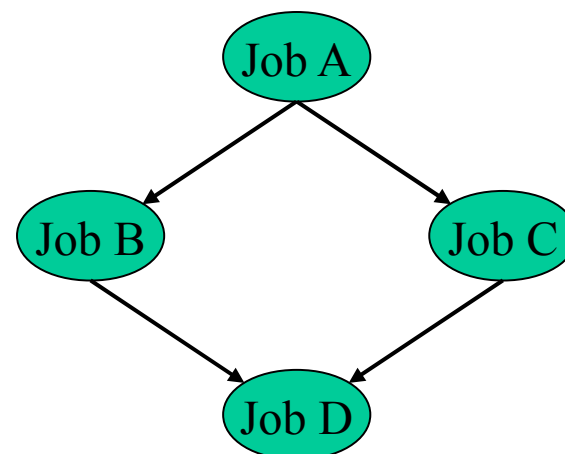
- A DAG is the **data structure** used by DAGMan to represent these dependencies.
- Each job is a “**node**” in the DAG.
- Each node can have any number of “parent or “children” nodes – as long as there are **no loops!**



# Defining a DAG

- A DAG is defined by a *.dag file*, listing each of its nodes and their dependencies:

```
# diamond.dag
Job A a.sub
Job B b.sub
Job C c.sub
Job D d.sub
Parent A Child B C
Parent B C Child D
```



- each node will run the Condor job specified by its accompanying Condor submit file

# Submitting a DAG

- To start your DAG, just run *condor\_submit\_dag* with your .dag file, and Condor will start a personal DAGMan daemon which to begin running your jobs:

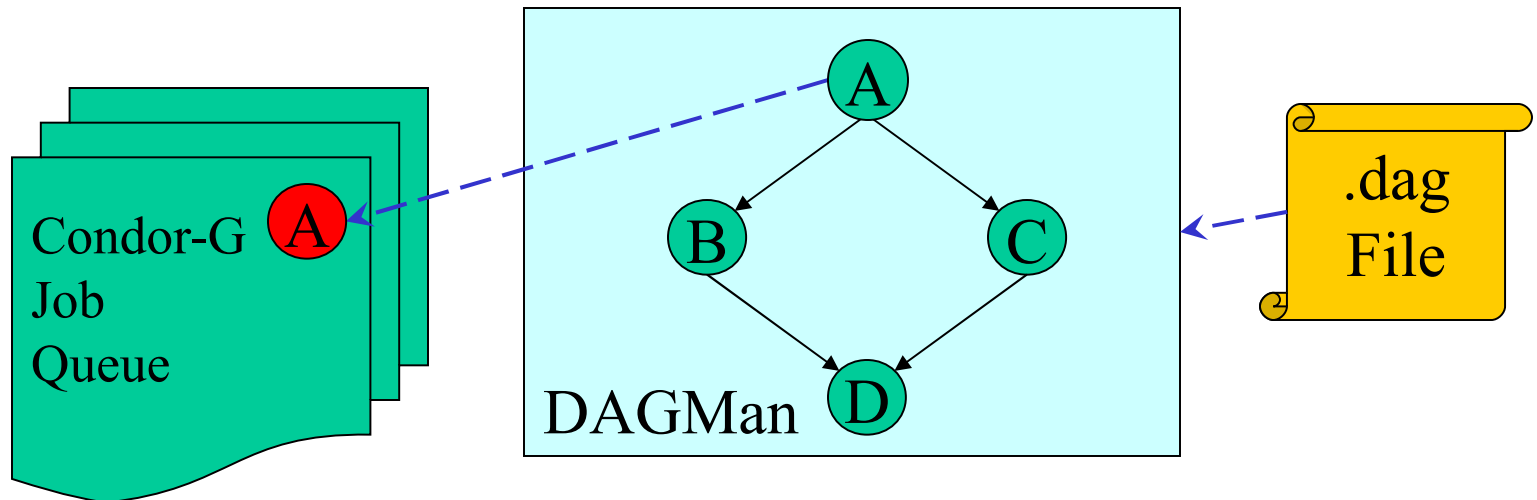
```
% condor_submit_dag diamond.dag
```

- `condor_submit_dag` submits a Scheduler Universe Job with DAGMan as the executable.
- Thus the DAGMan daemon itself *runs as a Condor job*, so you don't have to baby-sit it.



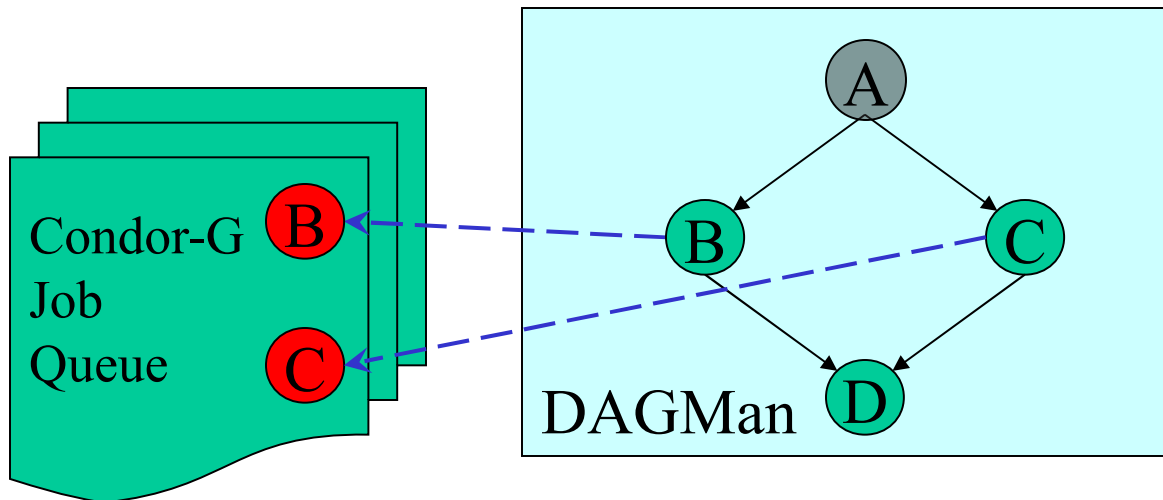
# Running a DAG

- DAGMan acts as a “meta-scheduler”, managing the submission of your jobs to Condor-G based on the DAG dependencies.



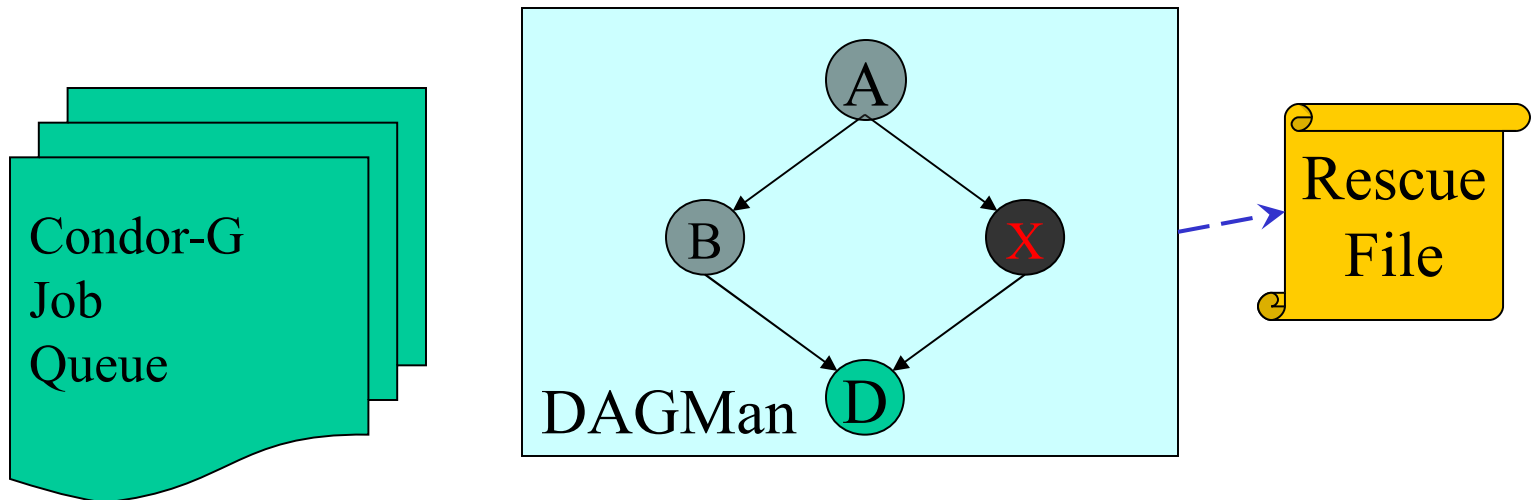
# Running a DAG (cont'd)

- DAGMan holds & submits jobs to the Condor-G queue at the appropriate times.



# Running a DAG (cont'd)

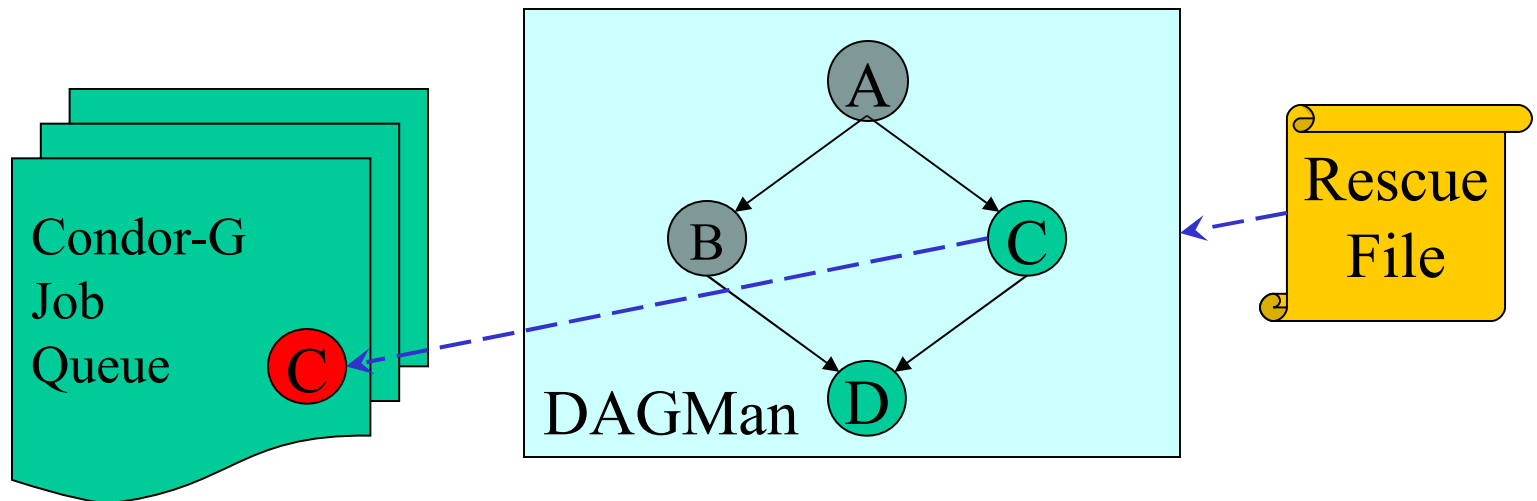
- In case of a job failure, DAGMan continues until it can no longer make progress, and then creates a *“rescue” file* with the current state of the DAG.



# Recovering a DAG

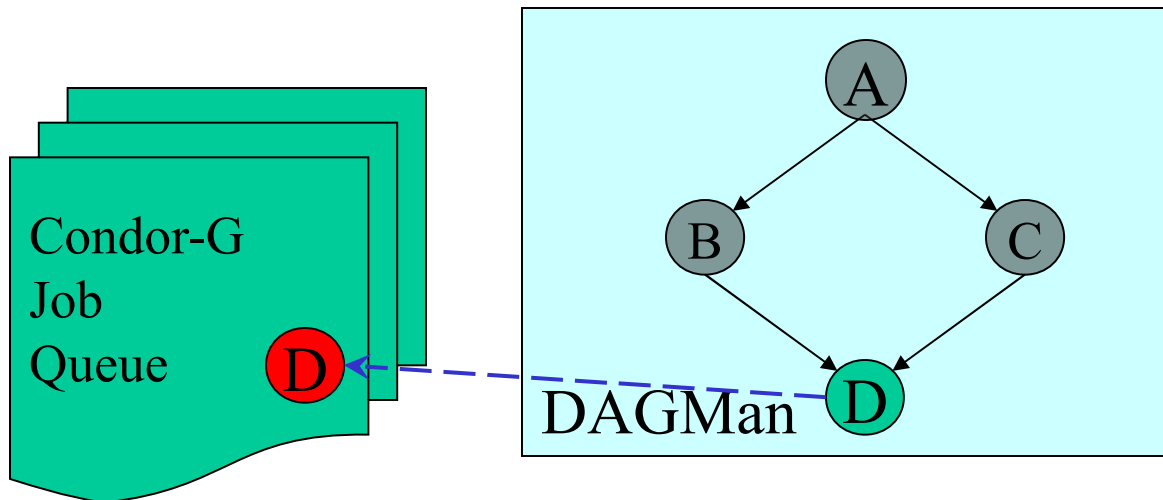
## -- fault tolerance

- Once the failed job is ready to be re-run, the rescue file can be used to restore the prior state of the DAG.



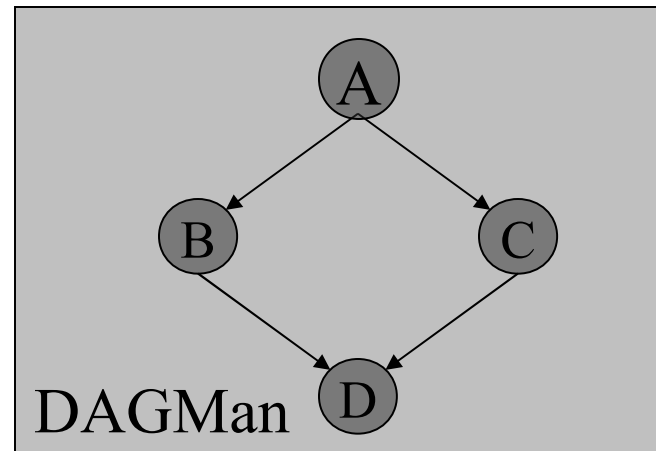
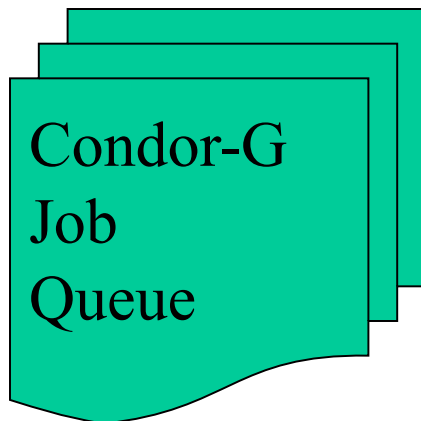
# Recovering a DAG (cont'd)

- Once that job completes, DAGMan will continue the DAG as if the failure never happened.



# Finishing a DAG

- Once the DAG is complete, the DAGMan job itself is finished, and exits.

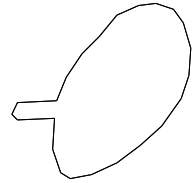


# We have seen how Condor:

- ... monitors submitted jobs and reports progress
- ... implements your policy on the execution order of the jobs
- ... keeps a log of your job activities

# Long jobs: if my jobs run for weeks

...



- What happens to my job when
  - a machine is shut down
  - there is a network outage, or
  - another job with higher priority preempts it?
  
- Do I lose all of those hours or days of computation time??
  
- What happens when they get pre-empted?
  
- *How can I add fault tolerance to my jobs?*



# Condor's Standard Universe to the rescue!

- Condor can support various combinations of features/environments in different “Universes”
- Different Universes provide different functionalities to your job:
  - Vanilla: Run any serial job
  - Scheduler: Plug in a scheduler
  - Standard: Support for transparent process checkpoint and restart

**provides two important services to your job:  
process checkpoint  
remote system calls.**

# Process Checkpointing

- Condor's process checkpointing mechanism *saves the entire state of a process into a checkpoint file*
  - Memory, CPU, I/O, etc.
- The process can then be *restarted* from the point it left off
- Typically no changes to your job's source code needed—however, your job must be relinked with Condor's Standard Universe support library

# OSG & job submissions

- OSG sites present interfaces allowing remotely submitted jobs to be accepted, queued and executed locally.
- OSG supports the Condor-G job submission client which interfaces to either the pre-web service or web services GRAM Globus interface at the executing site.
- Job managers at the backend of the GRAM gatekeeper support job execution by local Condor, LSF, PBS, or SGE batch systems.

## Acknowledgments:

This presentation based on:

Grid Resources and Job Management



**Open Science Grid**

Jaime Frey and Becky Gietzel

Condor Project

U. Wisconsin-Madison

---