the globus project™
www.globus.org

# GRAM: Grid Resource Allocation & Management

## Globus Toolkit™ Developer Tutorial


The Globus Project™

Argonne National Laboratory
USC Information Sciences Institute

http://www.globus.org/

# Components of Resource Management

- Resource Specification Language (RSL) is used to communicate requirements
- The Globus Resource Allocation Manager (GRAM) API allows programs to be started on remote resources
- A layered architecture allows application-specific resource brokers and co-allocators (e.g. DUROC) to be defined in terms of GRAM services

# GRAM is responsible for

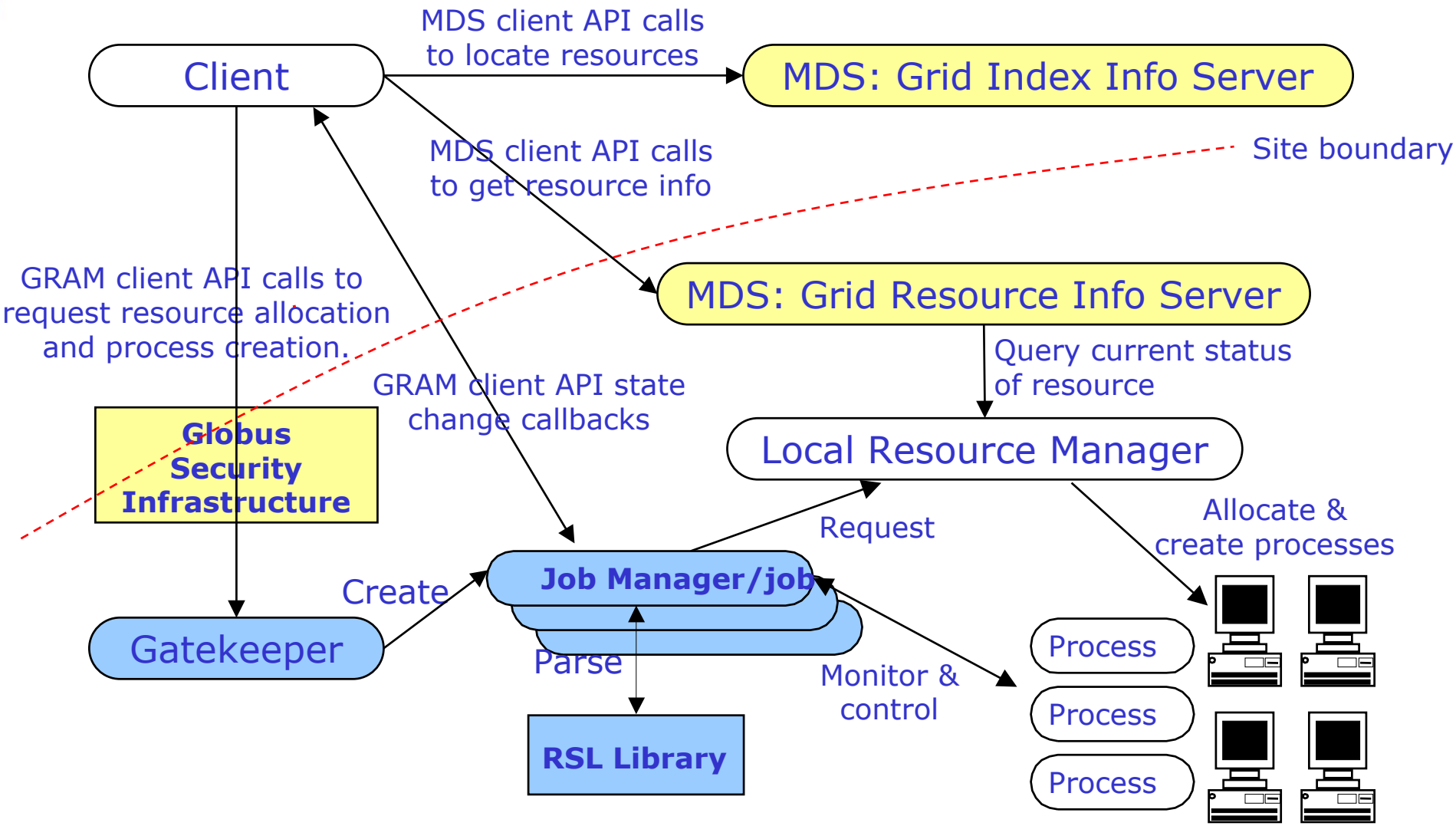
- Parsing and processing the RSL specifications
- Enabling remote monitoring and managing of jobs
- Updating MDS with information regarding the availability of the resources it manages

# Functionalities of GRAM

- GRAM allows you to run jobs remotely
- Provides an API for
  - Submitting
  - Monitoring
  - Terminating

  your job

# GRAM Components



# How GRAM works?

- To run a job remotely,
  - a GRAM gatekeeper server must be running on a remote computer, listening at port 2119
  - The application must be compiled on that remote machine
- The execution begins:
  - When a GRAM user application runs on the client machine
  - Sending a job request to the remote site
  - A job is submitted using the client API and creating an RSL specification
- Job request contains:
  - Executable, stdin, stdout
  - Name and port of the remote computer

# How GRAM works?

- The request is sent to the GRAM gatekeeper server of the remote computer which:
  - Checks globus certificate
  - Creates a job manager for the job
- The job manager:
  - Parses RSL
  - Starts and monitors the remote program
  - Communicates state changes to the client
  - Terminates when the job terminates

# Components of GRAM

- Job request – a request to gatekeeper to create one or more job processes, expressed in the RSL
- This request guides:
  - **Resource selection** (when and where to create the job processes)
  - **Job process creation** (what job processes to create)
  - **Job control** (how the processes should execute)

# Components of GRAM

- Gatekeeper – a process, running as root, which begins the process of handling allocation requests
- Its tasks:
  - Mutually authenticates with the client
  - Maps the requestor to a local user
  - Starts a job manager on the local host as the local user
  - Passes the allocation arguments to the newly created job manager

# Components of GRAM

- Job Manager – one process for each job, to fulfill every request submitted to the gatekeeper
- Its tasks:
  - Starts the job on the local system (e.g. by any local resource manager like Condor, SGE, etc.)
  - Handles all further communication with the client

# Two Components of the Job Manager

- Common Component
  - Translates messages received from the gatekeeper and client into an internal API
  - Translates callback requests from the machine specific components into messages to the application manager
- Machine-Specific Component
  - Implements the internal API in the local environment that includes
    - > Calls to the local system
    - > Message to the resource monitor
    - > Inquiries to the MDS

# Resource Management APIs

- Globus Toolkit has APIs for RSL, GRAM, and DUROC:
  - globus_rsl
  - globus_gram_client
  - globus_gram_myjob
  - globus_duroc_control
  - globus_duroc_runtime

# Resource Specification Language

- Much of the power of GRAM is in the RSL
- Common language for specifying job requests
  - GRAM service translates this common language into scheduler specific language
- GRAM service constrains RSL to a conjunction of (attribute=value) pairs
  - E.g. &(executable="/bin/ls")(arguments="-l")
- GRAM service understands a well defined set of attributes

# globus_rsl

- Module for manipulating RSL expressions
  - Parse an RSL string into a data structure
  - Functions to manipulate the data structure
  - Unparse the data structure into a string
- Can be used to assist in writing brokers or filters which refine an RSL specification

# globus_rsl

- Contains three main parts:
  - Information about the application program
  - Information about the remote computer
  - Control information (e.g. job_state_mask)

# RSL Attributes for the Program

- (executable=string)
  - Program to run
  - A file path (absolute or relative) or URL
- (directory=string)
  - Directory in which to run (default is $HOME)
- (arguments=arg1 arg2 arg3...)
  - List of string arguments to program
- (environment=(E1 v1)(E2 v2))
  - List of environment variable name/value pairs

# RSL Attributes for the Program

- (stdin=string)
  - Stdin for program
  - A file path (absolute or relative) or URL
- (stdout=string)
  - Stdout for program
  - A file path (absolute or relative) or URL
- (stderr=string)
  - Stderr for program
  - A file path (absolute or relative) or URL

# RSL Attributes for Control

- (count=integer)
  - Number of processes to run (default is 1)
- (hostCount=integer)
  - On SMP multi-computers, number of nodes to distribute the "count" processes across
- (project=string)
  - Project (account) against which to charge
- (queue=string)
  - Queue into which to submit job

# RSL Attributes for Control

- (maxWallTime=integer)
  - Maximum wall clock runtime in minutes
- (maxCpuTime=integer)
  - Maximum CPU runtime in minutes

# RSL Attributes for Control

- (maxMemory=integer)
  - Maximum amount of memory for each process in megabytes
- (minMemory=integer)
  - Minimum amount of memory for each process in megabytes

# RSL Attributes for Control

- (jobType=value)
  - Value is one of “mpi”, “single”, “multiple”, or “condor”
    - > mpi: Run the program using “mpirun -np <count>”
    - > single: Only run a single instance of the program, and let the program start the other count-1 processes.
    - > multiple: Start <count> instances of the program using the appropriate scheduler mechanism
    - > condor: Start a <count> Condor processes running in “standard universe”

# RSL Attributes for Control

- (gramMyjob=value)
  - Value is one of “collective”, “independent”
  - Defines how the globus_gram_myjob library will operate on the <count> processes
    - collective: Treat all <count> processes as part of a single job
    - independent: Treat each of the <count> processes as an independent uniprocessor job
- (dryRun=true)
  - Do not actually run job

# Constraints: "&"

- For example:

  "Create 5-10 instances of myprog, each on a machine with at least 64 MB memory that is available to me for 4 hours"

  ```
  & (count>=5) (count<=10)
     (max_time=240) (memory>=64)
     (executable=myprog)
  ```

# Disjunction: "|"

- For example:
- Create 5 instances of myprog on a machine that has at least 64MB of memory, or 10 instances on a machine with at least 32MB of memory

```
& (executable=myprog)
   ( | (&(count=5)(memory>=64))
       (&(count=10)(memory>=32)))
```

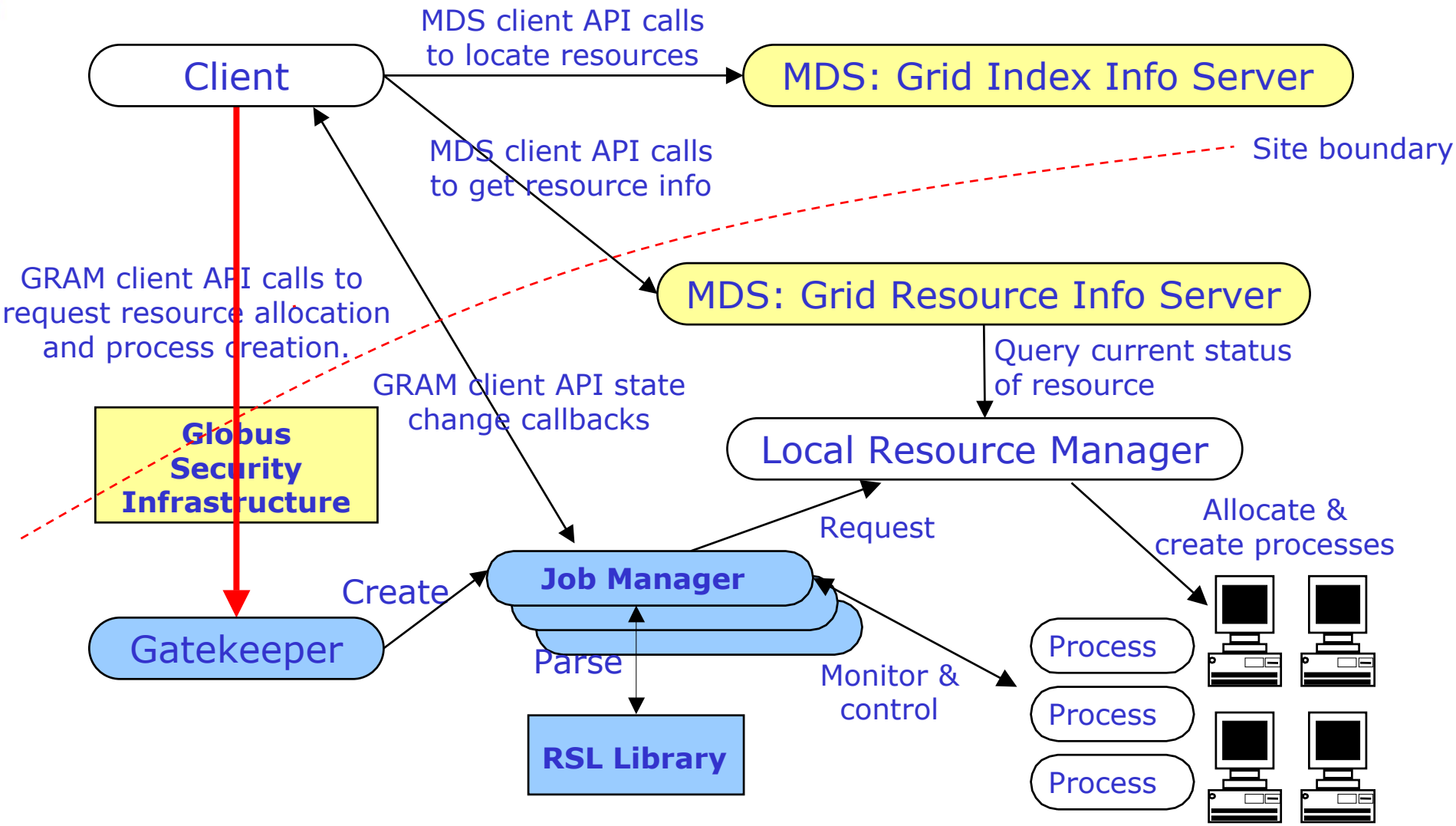
# Resource Management APIs

- Globus Toolkit has APIs for RSL, GRAM, and DUROC:
  - globus_rsl
  - globus_gram_client
  - globus_gram_myjob
  - globus_duroc_control
  - globus_duroc_runtime

# globus_gram_client

- globus_gram_client_job_request()
  - Submit a job to a remote resource
  - Input:
    - Resource manager contact string
    - RSL specifying the job to be run
    - Callback contact string, for notification
  - Output:
    - Job contact string

# GRAM Components



# Finding The Gatekeeper

- globus_gram_client_job_request() requires a resource manager contact string to find the gatekeeper

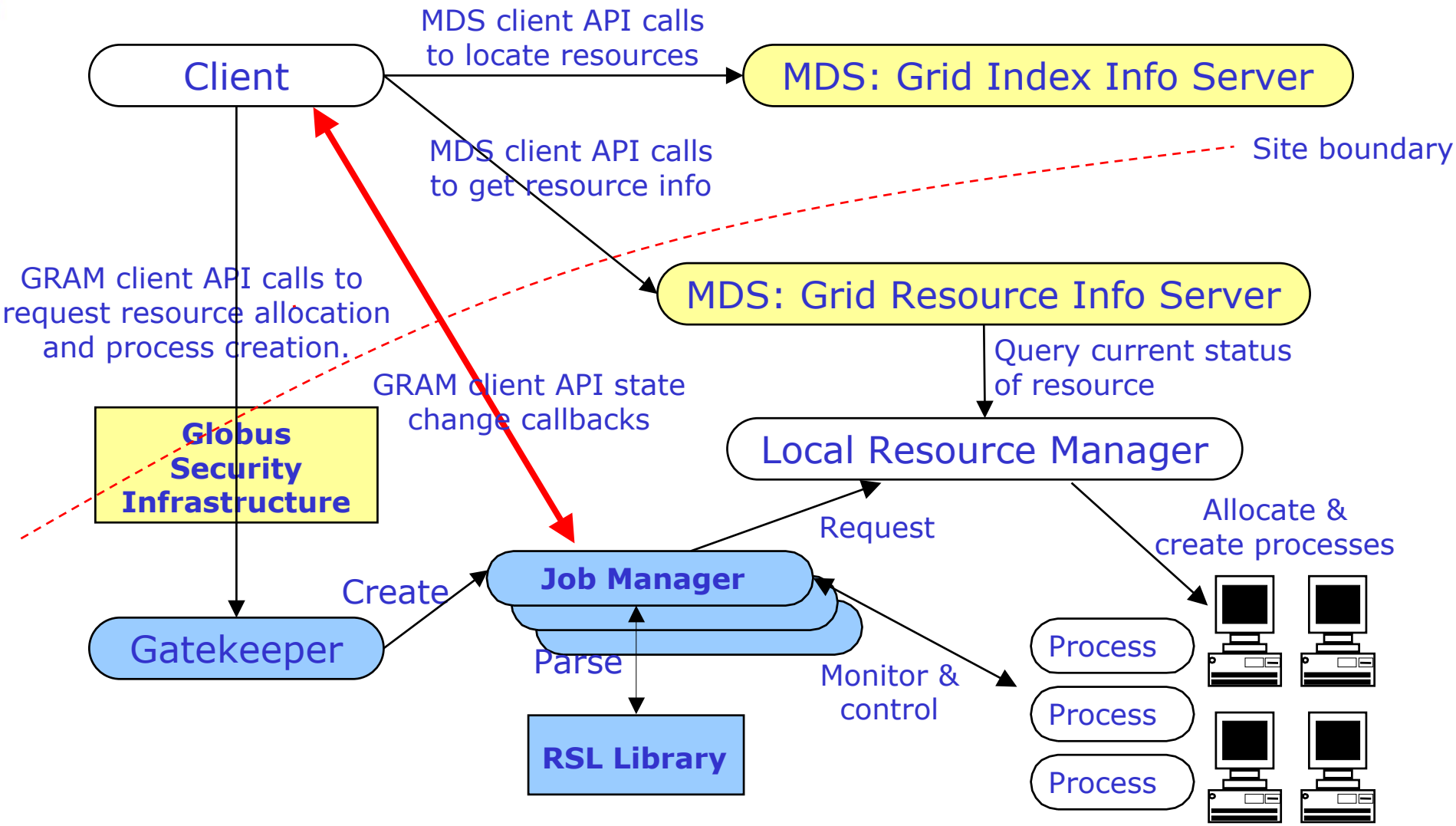
  hostname[:port][/service][:subject]

  - hostname – host of gatekeeper
    - > required
  - port – port on which gatekeeper is listening
    - > defaults to well known port = gsigatekeeper = 2119
  - service – gatekeeper service to invoke
    - > defaults to "jobmanager"
  - subject – security subject name of gatekeeper
    - > Defaults to standard host cert form: "…/cn=host/hostname"

# Job Contact

- globus_gram_client_job_request() returns a job contact string
  - Other globus_gram_client_*() functions use the job contact to find the right job manager to which requests are made
  - Job contact string can be passed between processes, even on different machines

# GRAM Components



# globus_gram_client

- globus_gram_client_job_status()
  - Check the status of the job
    - > UNSUBMITTED, PENDING, ACTIVE, FAILED, DONE, SUSPENDED
  - Can also get job status through callbacks
    - > globus_gram_client_callback_{allow,disallow,check}()
- globus_gram_client_job_cancel()
  - Cancel/kill a pending or active job

# globus_gram_client

- globus_gram_client_job_signal()
  - Controls the jobmanager
  - COMMIT_REQUEST*
    - > submit job
  - COMMIT_END*
    - > Cleanup job
  - COMMIT_EXTEND*
    - > Wait additional N seconds
  - * when jobs have “(two_phased=yes)”

# globus_gram_client

- globus_gram_client_job_signal(), continued
  - STDIO_UPDATE
    - > Allows client to submit an RSL that changes some I/O attributes of the job
      - stdout, stderr, stdout_position, stderr_position, remote_io_url
  - STDIO_SIZE
    - > verify that streamed I/O has been completely received
  - STOP_MANAGER
    - > Tells JM to exit, but leave the job running

# State Change Callbacks

- GRAM managed job can be in the states:
  - Unsubmitted, Pending, Active, Failed, Done, Suspended
- GRAM client can register for asynchronous state change callbacks
  - Registration can be done either:
    - during submission by globus_gram_client_job_request()
    - or later by any process, using the job contact globus_gram_client_job_callback_register()

# globus_gram_client

- globus_gram_client_callback_allow()
  globus_gram_client_callback_disallow()
  globus_gram_client_callback_check()
  - Create/destroy a client port to listen for asynchronous state change callbacks
  - Callback to local function on state change
- globus_gram_client_job_callback_register()
  globus_gram_client_job_callback_unregister()
  - Register with job manager to receive callbacks

# globus_gram_myjob

- When a set of processes in a single job startup, they may need to self organize
  - How many processes in the job?
  - What is my rank within the job?
  - Simple send/receive between job processes.
- This API is a minimal set of functions to allow this self organization
- This is a bootstrapping library. It is NOT meant to be a general purpose message passing library for use by applications

# DUROC Review

- Simultaneous allocation of a resource set
  - Handled via optimistic co-allocation based on free nodes or queue prediction
  - In the future, advance reservations will also be supported
- globusrun will co-allocate specific multi-requests
  - Uses a Globus component called the **Dynamically Updated Request Online Co-allocator** (DUROC)

# Multirequest: "+"

- A multirequest allows us to specify multiple resource needs, for example

```
+ (& (count=5)(memory>=64)
       (executable=p1))
  (&(network=atm) (executable=p2))
```

  - Execute 5 instances of p1 on a machine with at least 64M of memory
  - Execute p2 on a machine with an ATM connection
- Multirequests are central to co-allocation

# A Co-allocation Multirequest

```
+( & (resourceManagerContact=
     *** "flash.isi.edu:2119/jobmanager-
lsf:/O=Grid/…/CN=host/flash.isi.edu")
       (count=1)
       (label="subjob A")
       (executable= my_app1)
   )
   ( & (resourceManagerContact=
***"sp139.sdsc.edu:2119:/O=Grid/…/CN=host/sp097.sdsc.edu")
       (count=2)
       (label="subjob B")
       (executable=my_app2)
   )
```

Different resource managers

Different counts

Different executables

# RSL Attributes For DUROC

- (subjobStartType=value)
  - Alters the startup barrier mechanism
  - values are “strict-barrier”, “loose-barrier”, “no-barrier”
- (subjobCommsType=value)
  - values are “blocking-join” and “independent”
  - if value is set to “independent”, the subjob won’t be seen from the other subjobs when doing inter-subjob communication.

# RSL Attributes For DUROC

- (label=string)
  - Identifier for this subjob
- (resourceManagerContact=string) (resourceManagerName=string)
  - Resource manager to which to submit a subjob

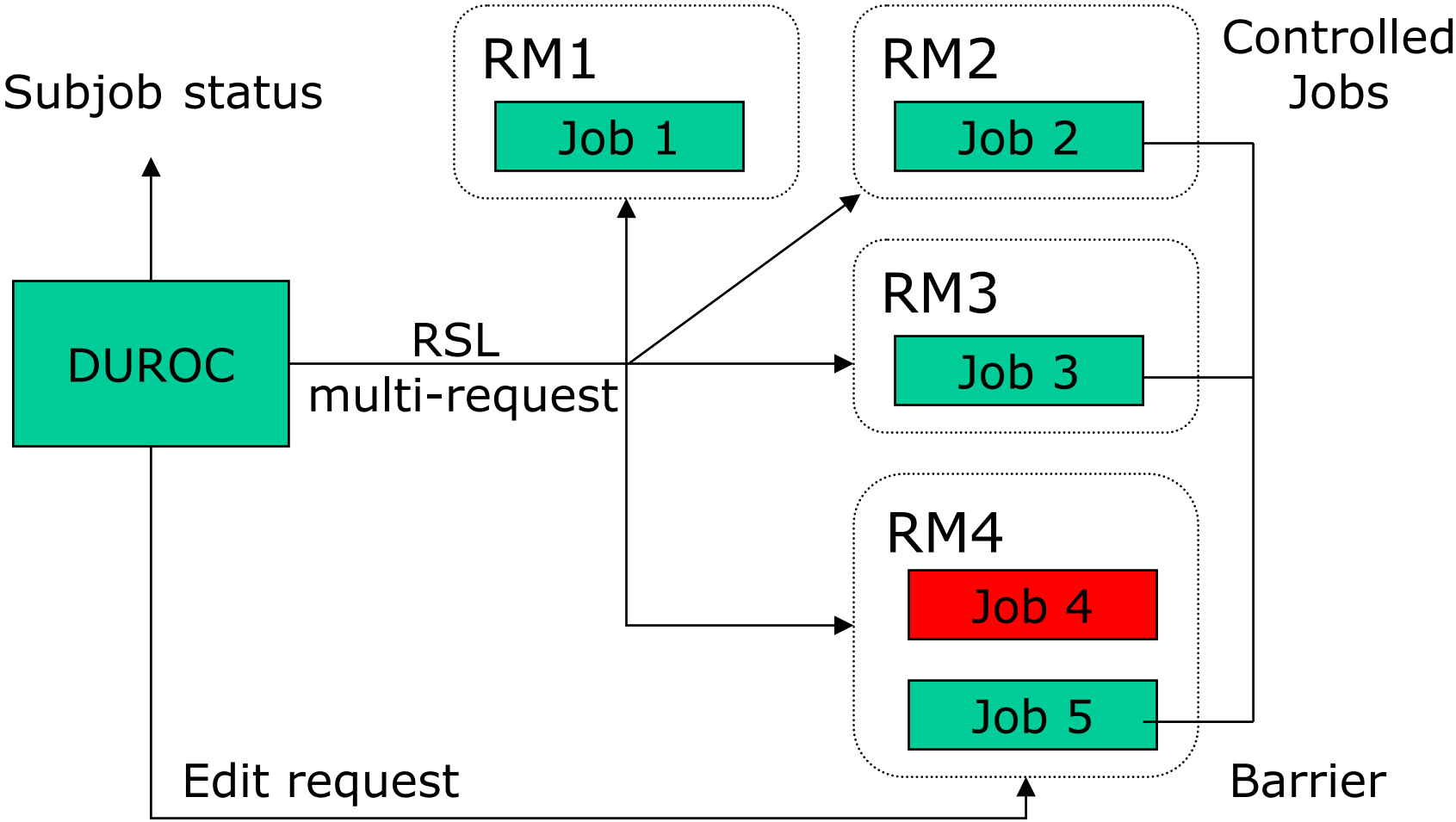
# globus_duroc_control Module

- Submit a multi-request
- Edit a pending request
  - Add new nodes, edit out failed nodes
- Commit to configuration
  - Delay to last possible minute
  - Barrier synchronization
- Initialize computation
- Monitor and control collection

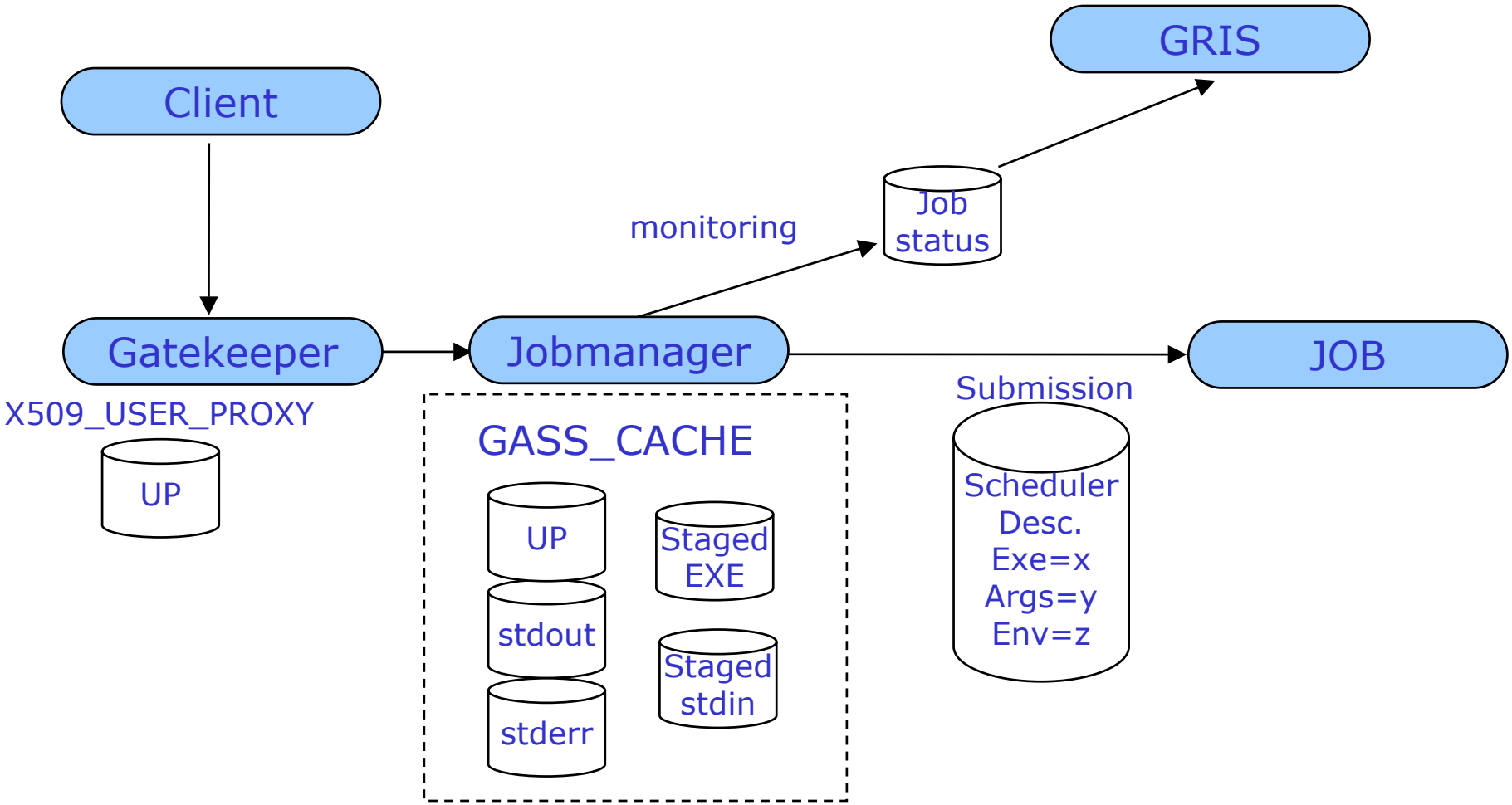
# globus_duroc_runtime Module

- globus_duroc_runtime_barrier()
  - All processes in DUROC job must call this
  - It will wait until the DUROC control module releases all processes from the barrier
- globus_duroc_runtime_inter_subjob_*()
  - Bootstrap library between subjobs
- globus_duroc_runtime_intra_subjob_*()
  - Bootstrap library within a subjob

# DUROC Architecture



# Job Manager Files



# Using Information for Resource Brokering

the globus project™
www.globus.org

"10 GFlops, EOS data,
20 Mb/sec -- for 20 mins"

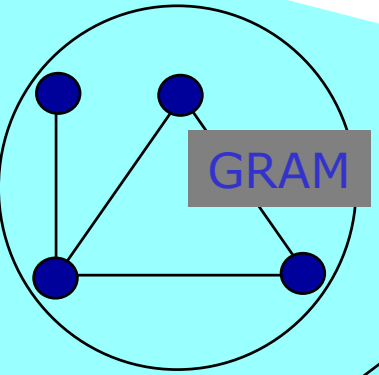
Info service:
location + selection

Resource Broker

"What computers?"
"What speed?"
"When available?"

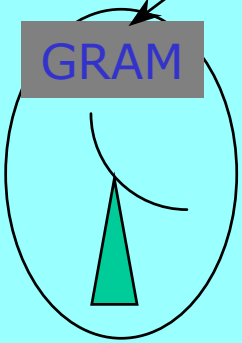
Metacomputing Directory Service

"20 Mb/sec"

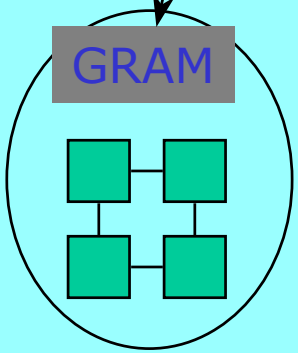


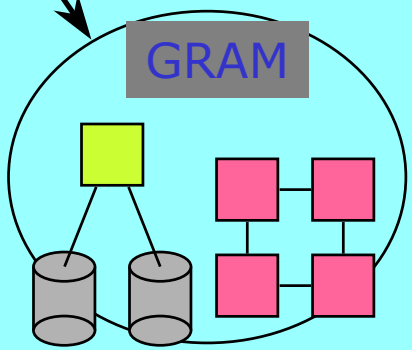

Globus Resource Allocation Managers

"50 processors + storage
from 10:20 to 10:40 pm"

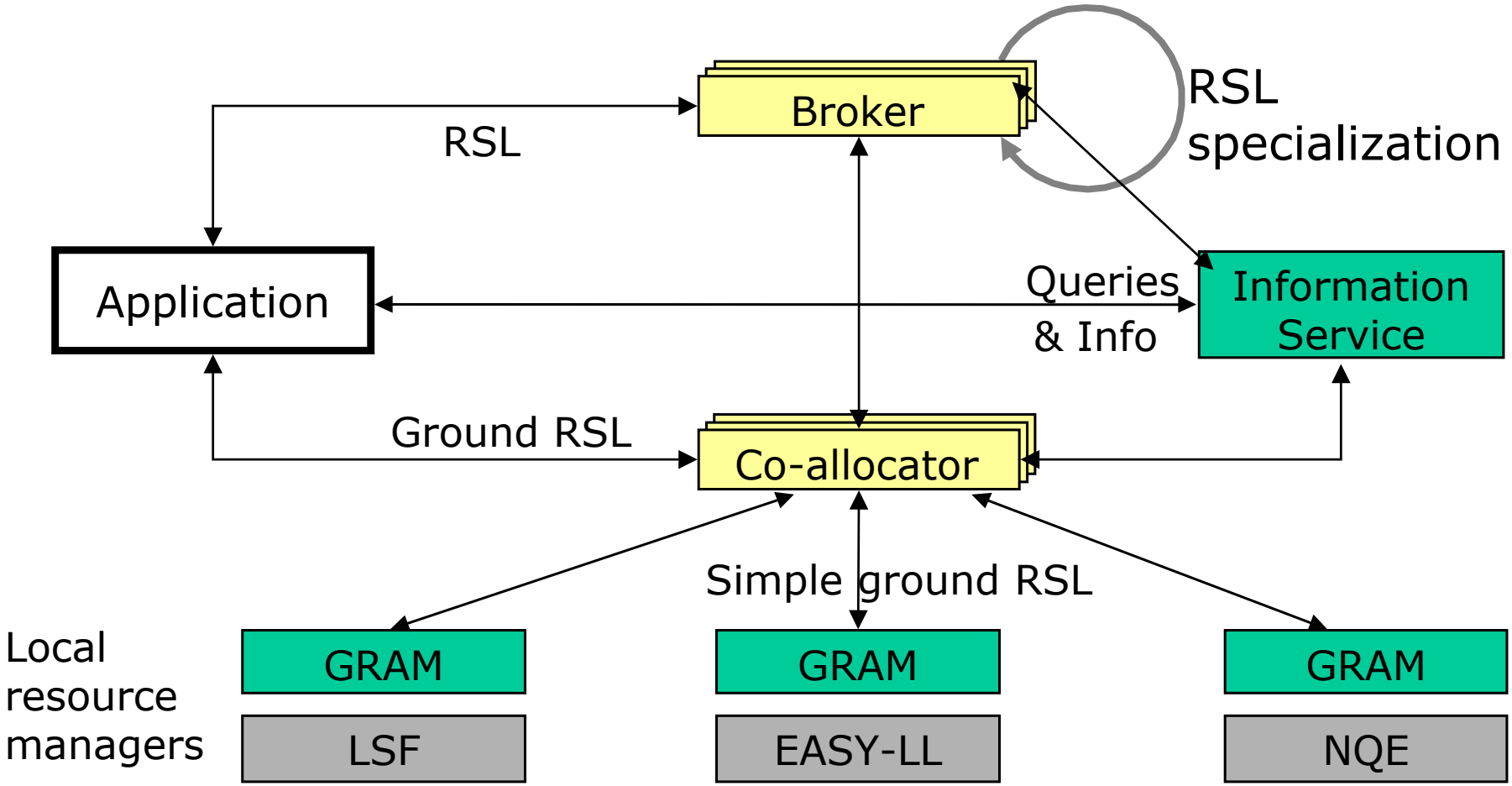



Fork
LSF
EASYLL
Condor
etc.



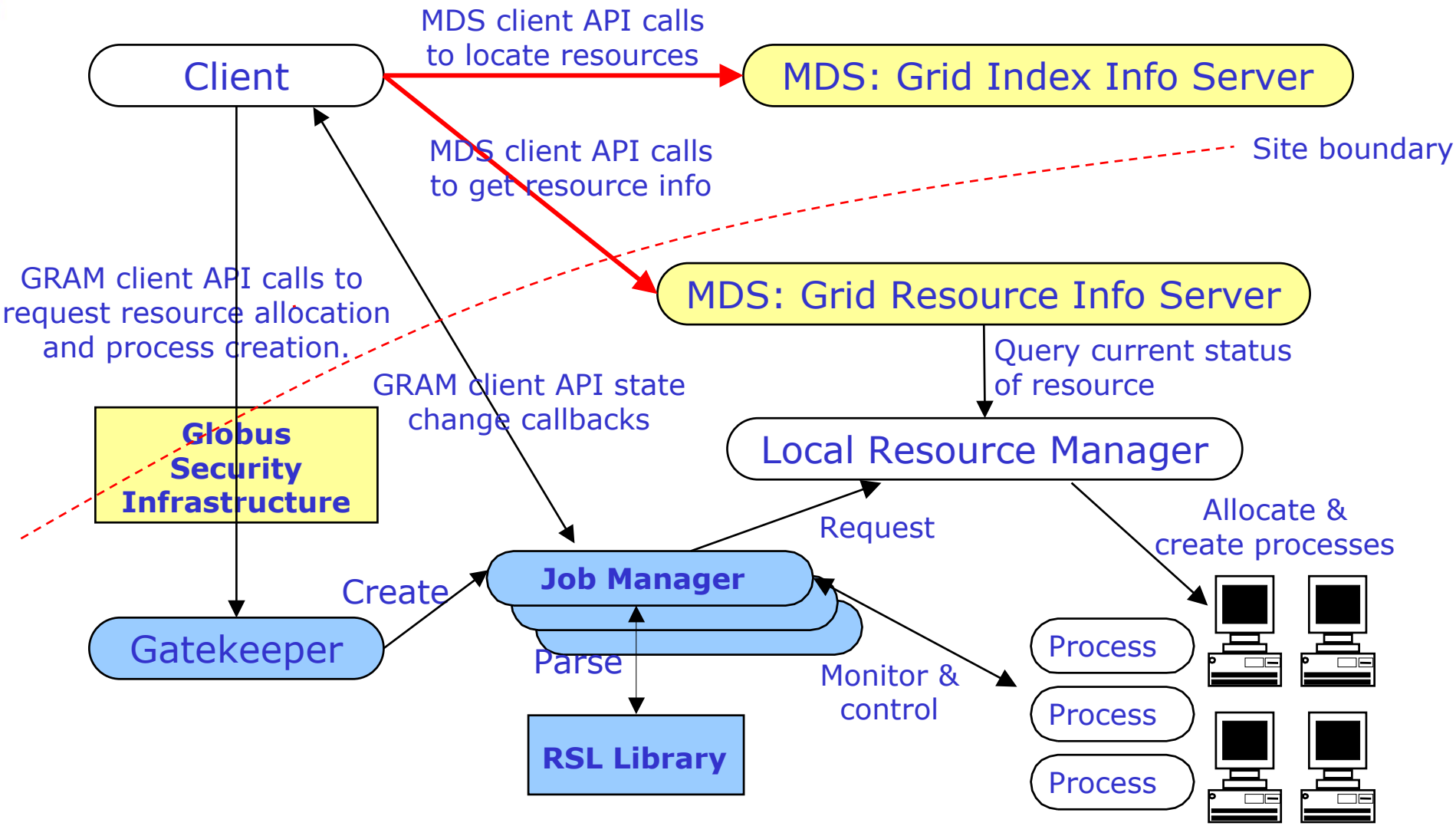
# Resource Management Architecture



# Resource Broker

- The Globus Toolkit does not include a resource broker or a metascheduler!
- It is the task of the user to access MDS (GIIS and GRIS) and select the remote site where the program should run

# GRAM Components



# Resource Broker

- The Globus team helped many people to build brokers using GRAM and MDS services
- Several brokers now exist:
  - Condor-G, DRM, PUNCH, Nimrod/G, Cactus, AppLeS,