CASTOR Tutorial

Part 2
Functional description

Outline

- Detailed view of the architecture
  - Lifecycle of a GET and a PUT request
- Description and status of the components
  - Main daemons
  - Diskserver related
  - Central services
  - Tape related
- Tape migration and recall
  - Workflow details

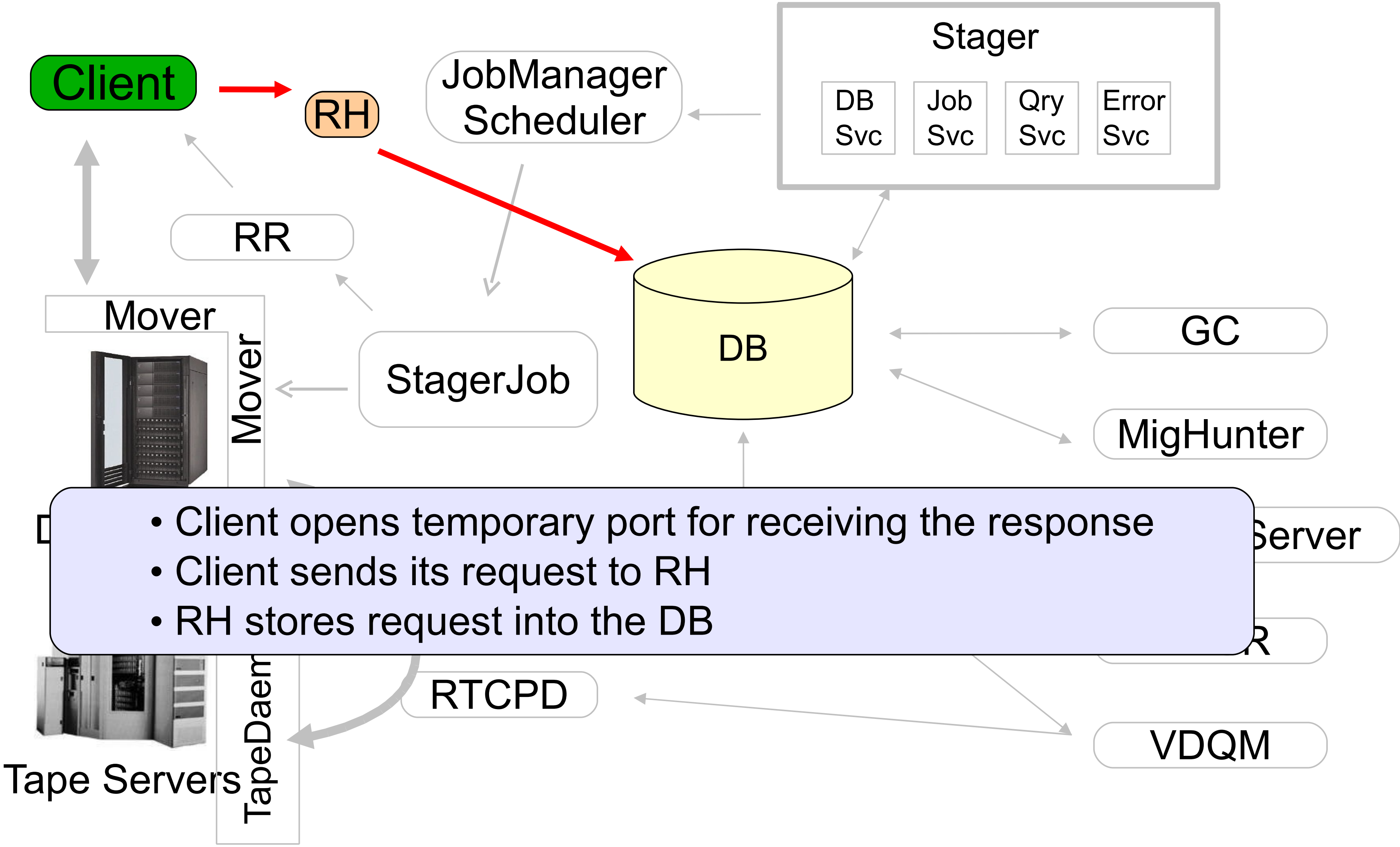
Outline

- Detailed view of the architecture
  - Lifecycle of a GET and a PUT request
- Description and status of the components
  - Main daemons
  - Diskserver related
  - Central services
  - Tape related
- Tape migration and recall
  - Workflow details

Lifecycle of a GET + recall

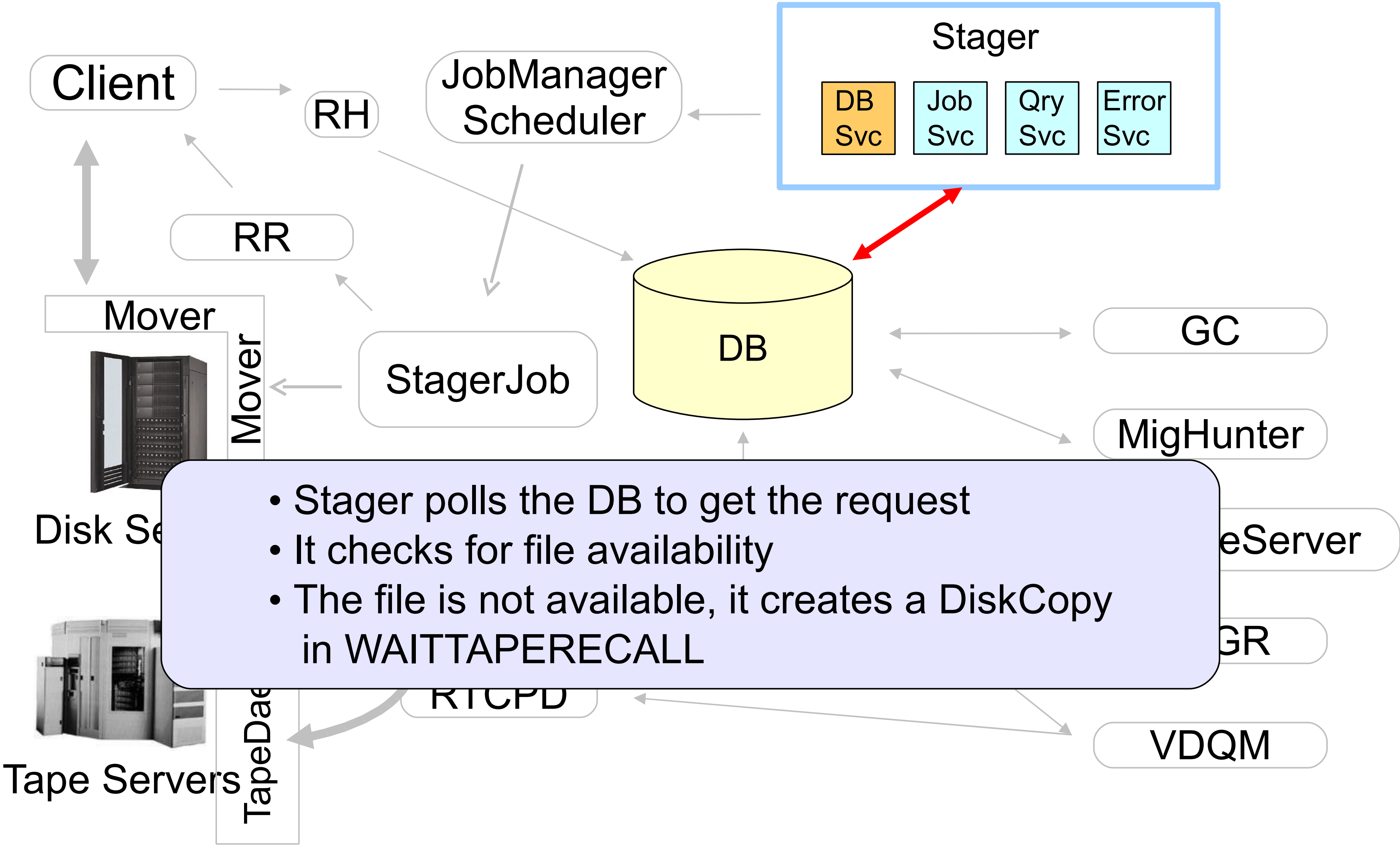
- Client connects to the RH
- RH stores the request into the db
- Stager polls the db and checks for file availability
- If the file is not available, the recall process is activated
- Once the file is available, stager asks the jobManager to schedule the access to the file
- The JobManager enters a job into LSF and babysits it
- After the job was started, llient gets a callback and can initiate the transfer
- The commandline is stager_get

stager_get (1)



- Client opens temporary port for receiving the response
- Client sends its request to RH
- RH stores request into the DB

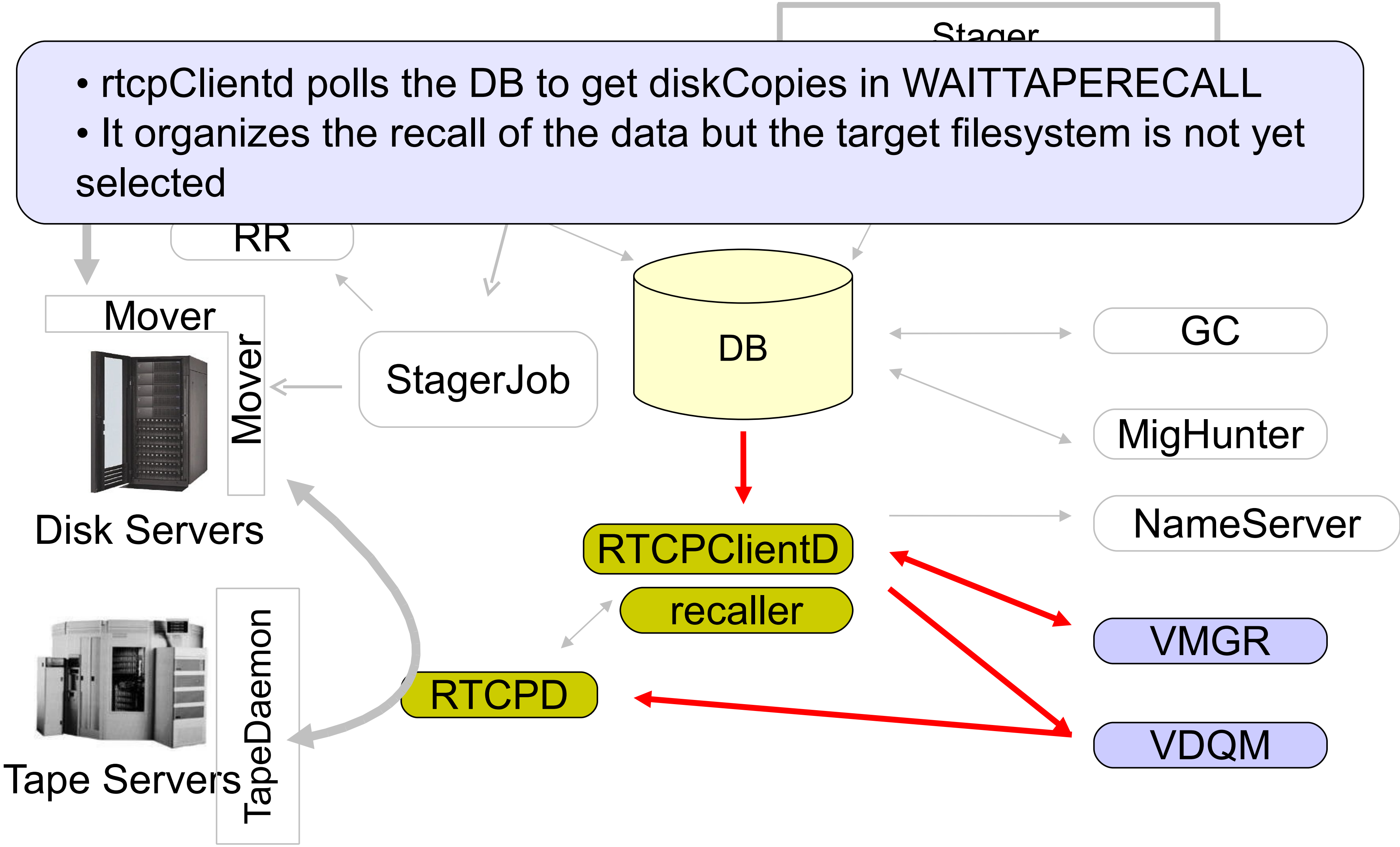
stager_get (2)



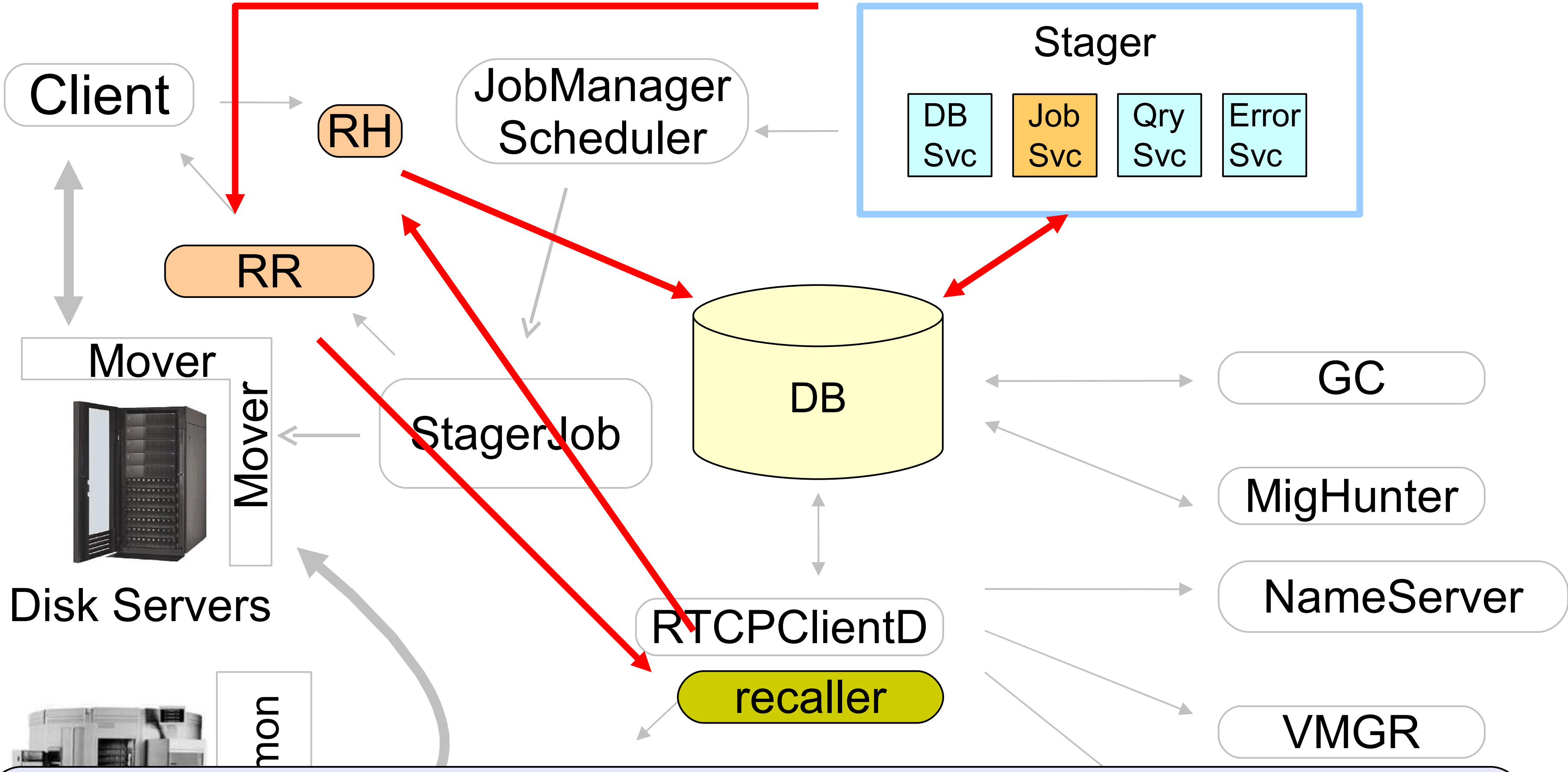
- Stager polls the DB to get the request
- It checks for file availability
- The file is not available, it creates a DiskCopy in WAITTAPERECALL

stager_get (3)

- rtcpClientd polls the DB to get diskCopies in WAITTAPERECALL
- It organizes the recall of the data but the target filesystem is not yet selected



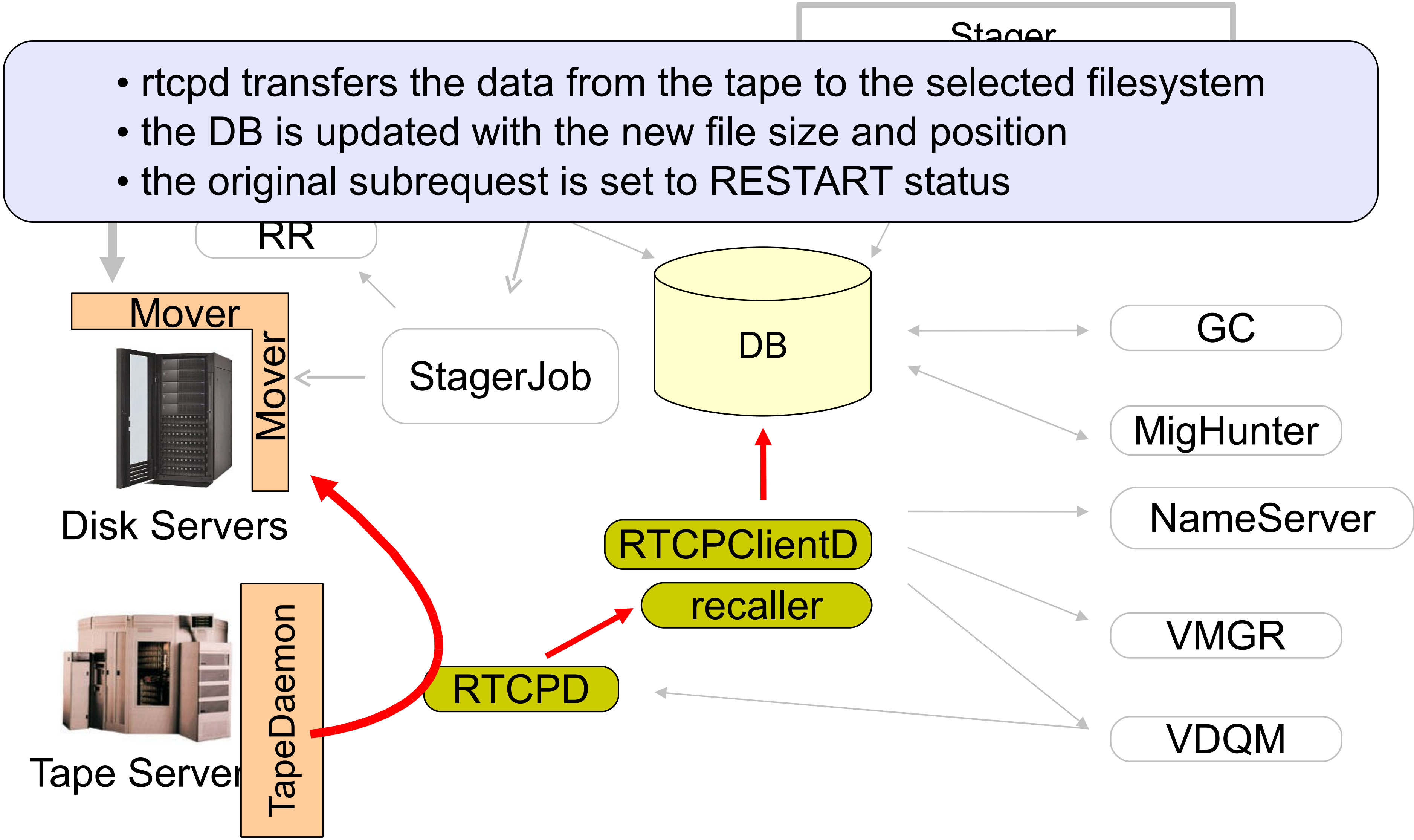
stager_get (4)



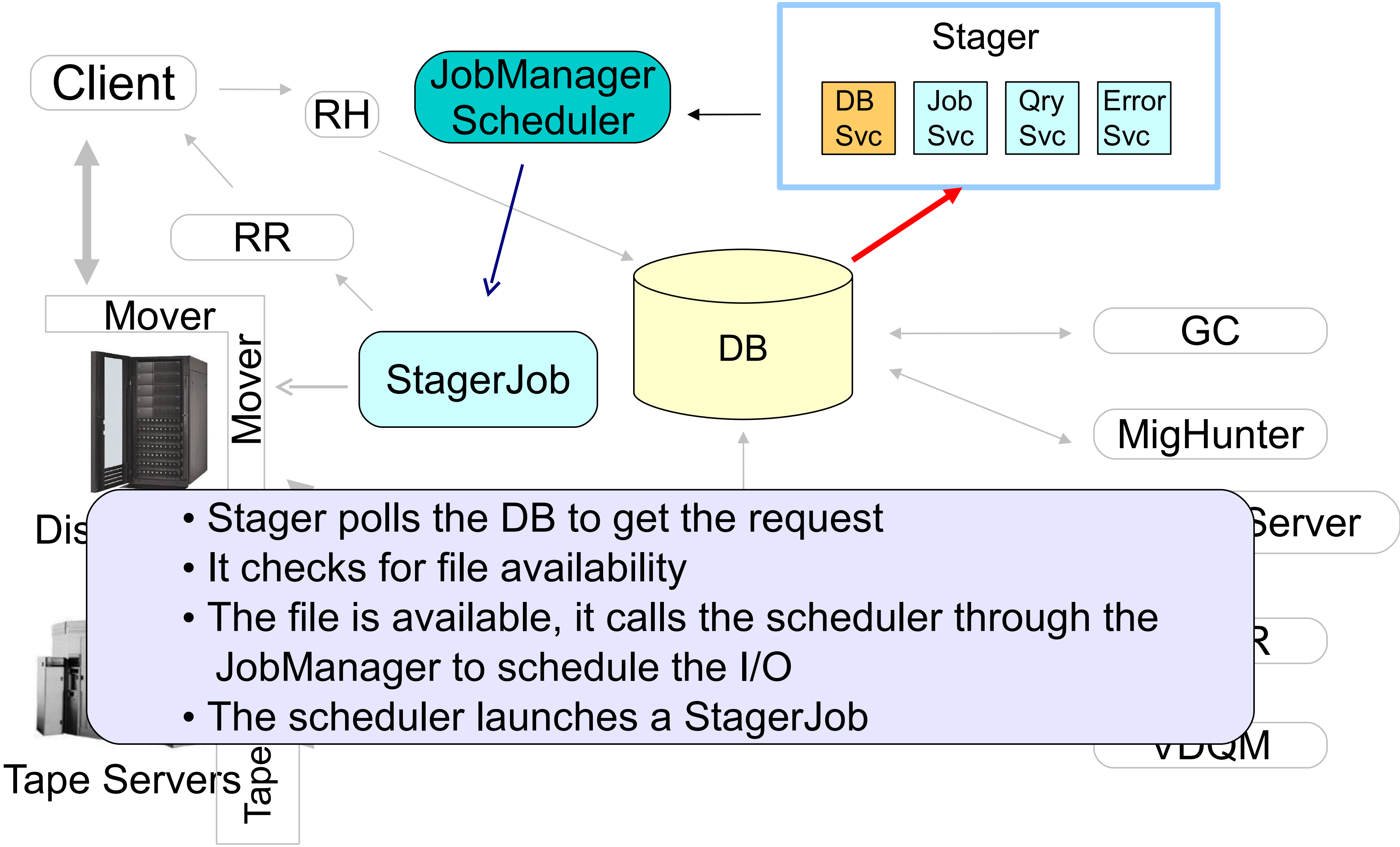
- recaller sends a request to the stager in order to know where to put the file
- the request goes through the usual way: Request Handler, DB, stager (job service), Request Replier

stager_get (5)

- rtcpd transfers the data from the tape to the selected filesystem
- the DB is updated with the new file size and position
- the original subrequest is set to RESTART status

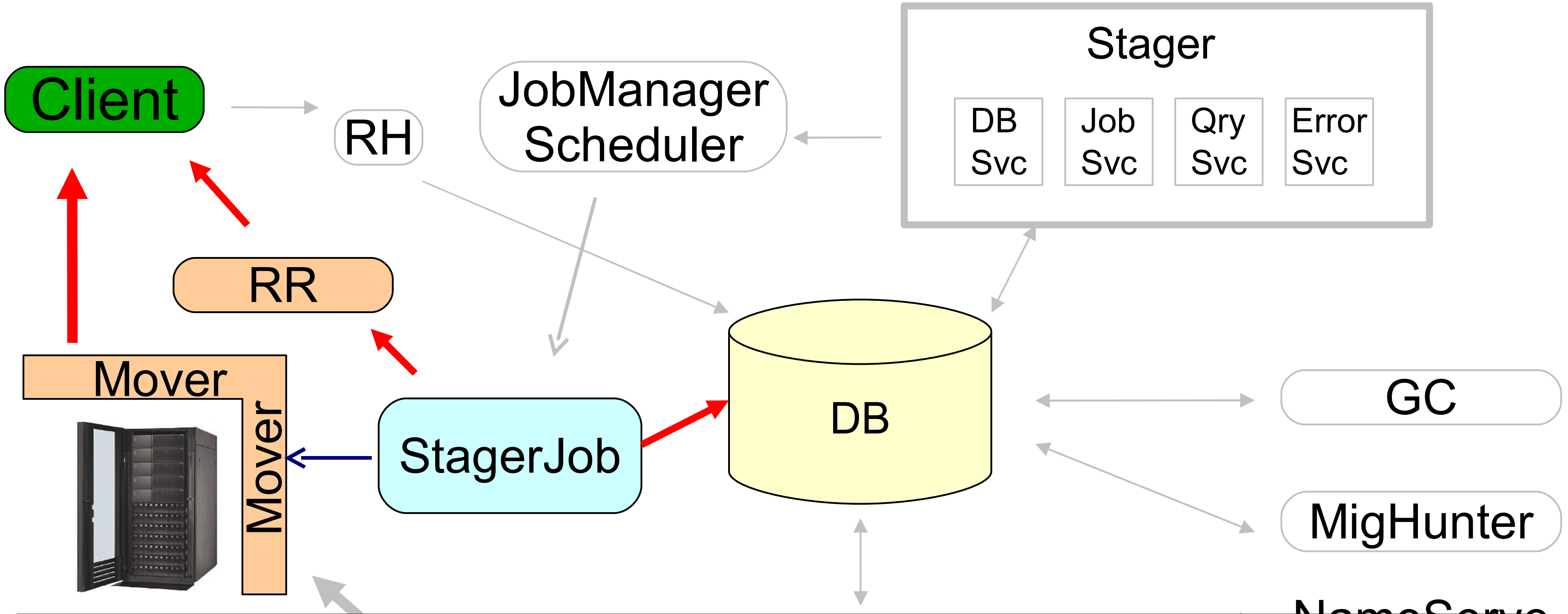


stager_get (6)



- Stager polls the DB to get the request
- It checks for file availability
- The file is available, it calls the scheduler through the JobManager to schedule the I/O
- The scheduler launches a StagerJob

stager_get (7)



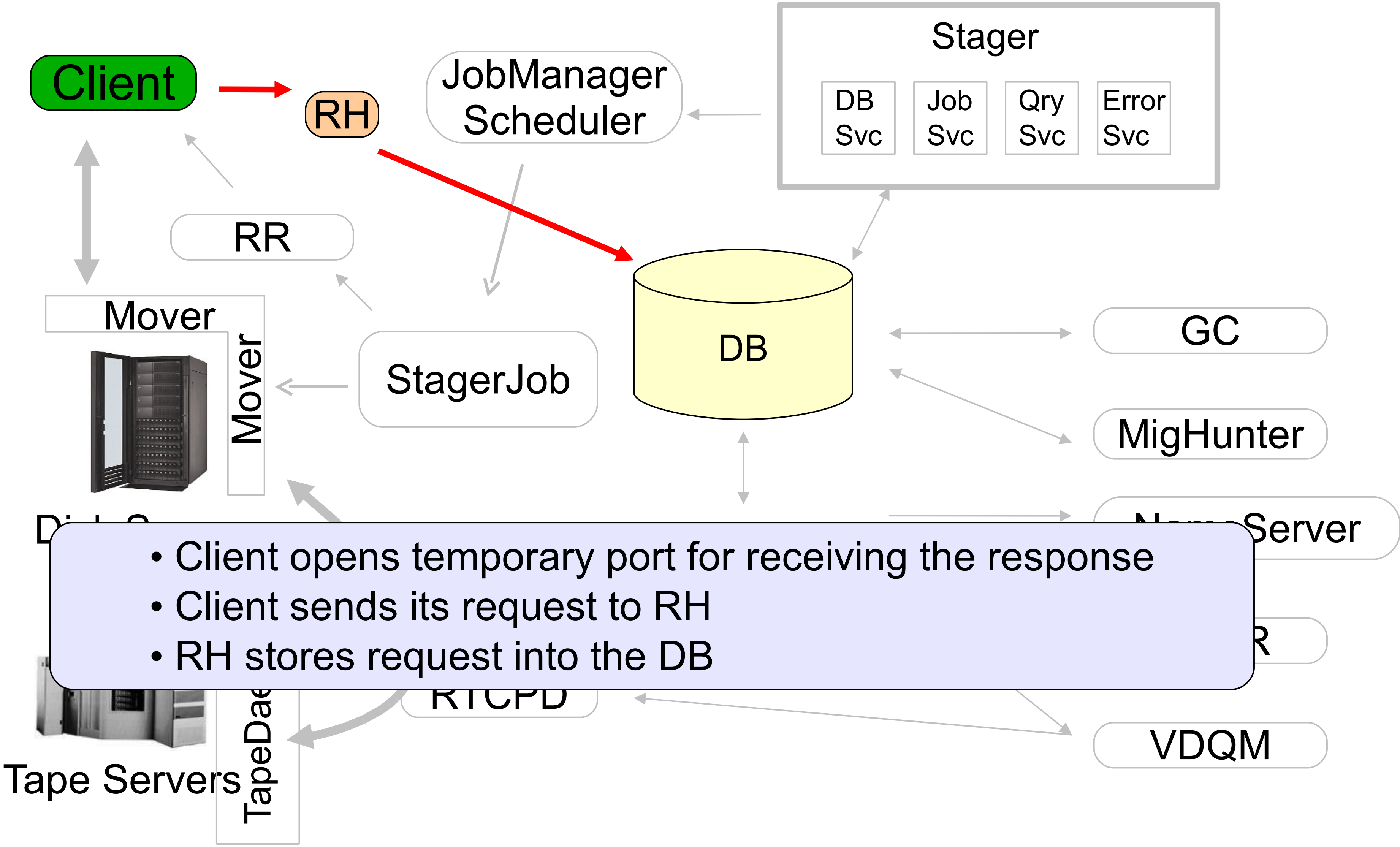
- the StagerJob launches the right mover corresponding to the client request
(note that the scheduler takes available movers into account)
- it answers to the client, giving to it the machine and port where to contact the mover
- data is transfered
- DB is updated

Lifecycle of a PUT + migration

- Client connects to the RH
- RH stores the request into the db
- Stager polls the db and ask the jobManager to scheduler the transfer
- The JobManager enters an LSF job and babysits it
- Once the job starts, the client gets a callback and can initiate the transfer
- After the transfer is completed, the stager is informed and migration to tape is triggered

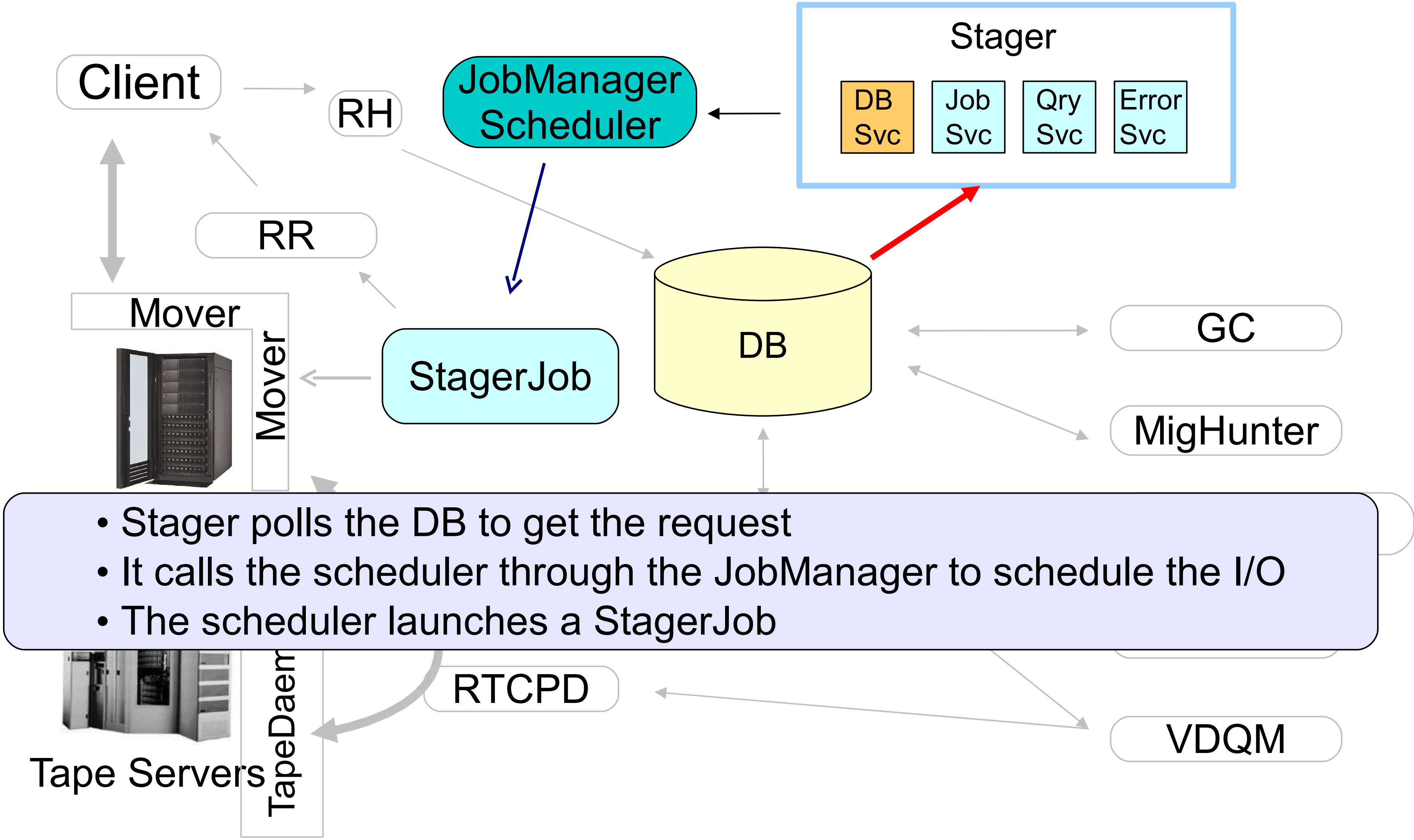
- The commandline is stager_put

stager_put (1)



- Client opens temporary port for receiving the response
- Client sends its request to RH
- RH stores request into the DB

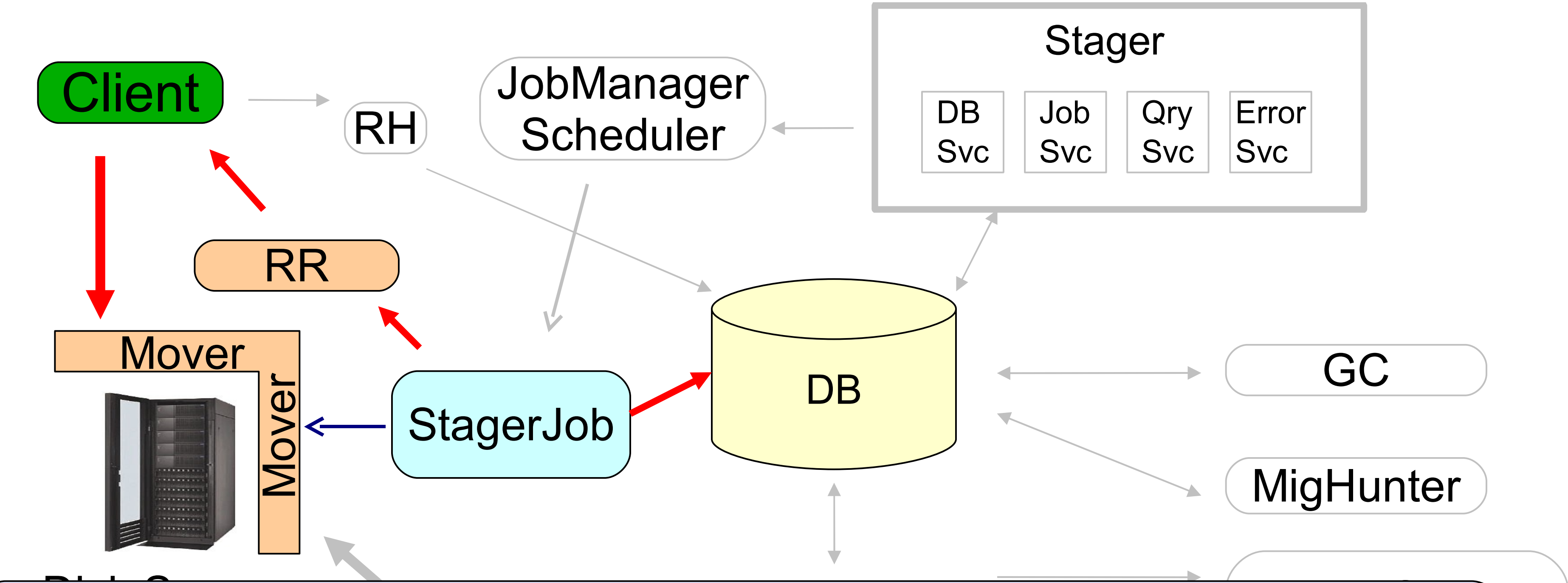
stager_put (2)



- Stager polls the DB to get the request
- It calls the scheduler through the JobManager to schedule the I/O
- The scheduler launches a StagerJob

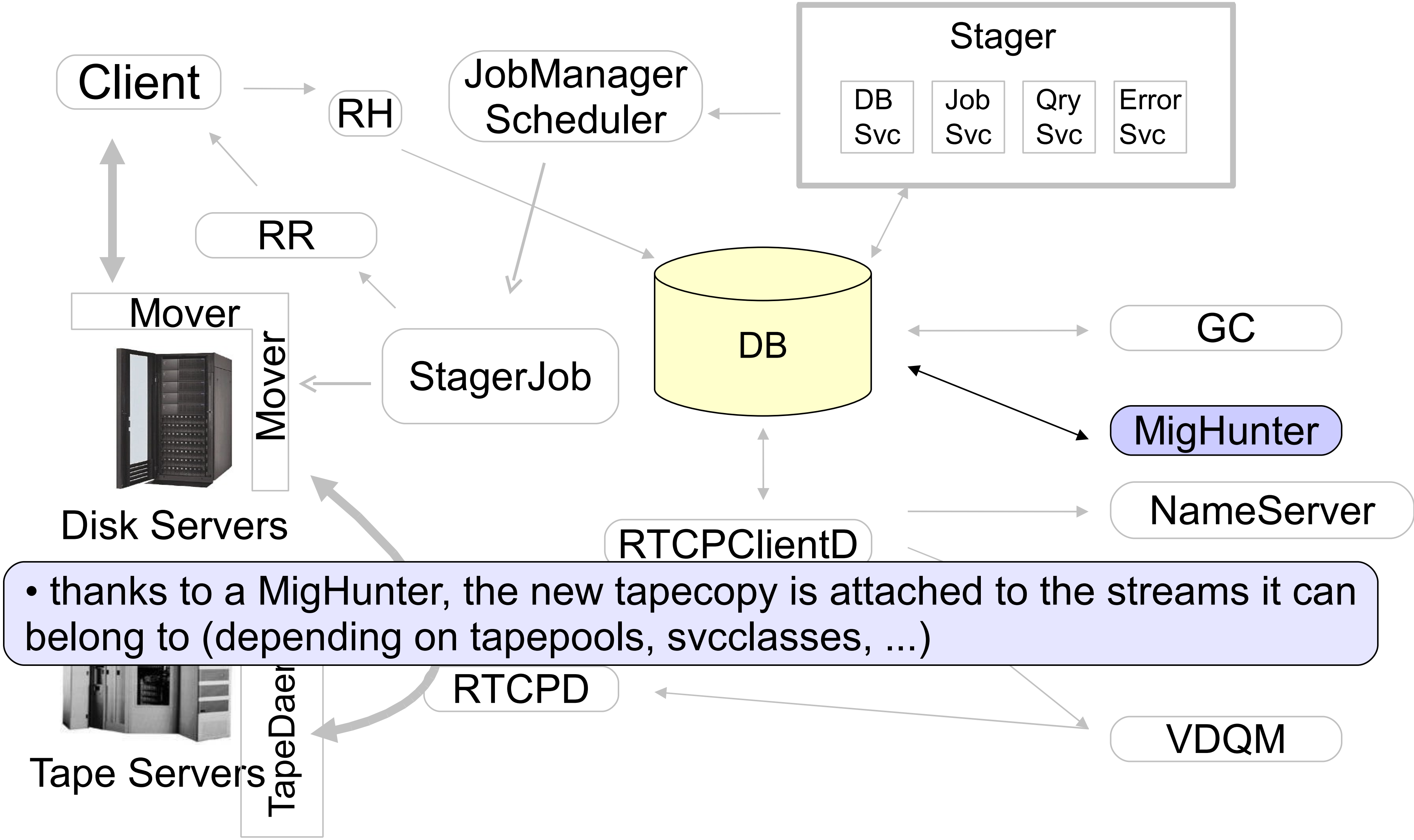
stager_put (3)

CERN IT Department



- the StagerJob launches the right mover (note that the scheduler takes available movers into account)
- it answers to the client, giving the machine and port where to contact the mover
- data is transferred
- DB is updated with the file size and the diskcopy is set in CANBEMIGR and one or many TapeCopies are created

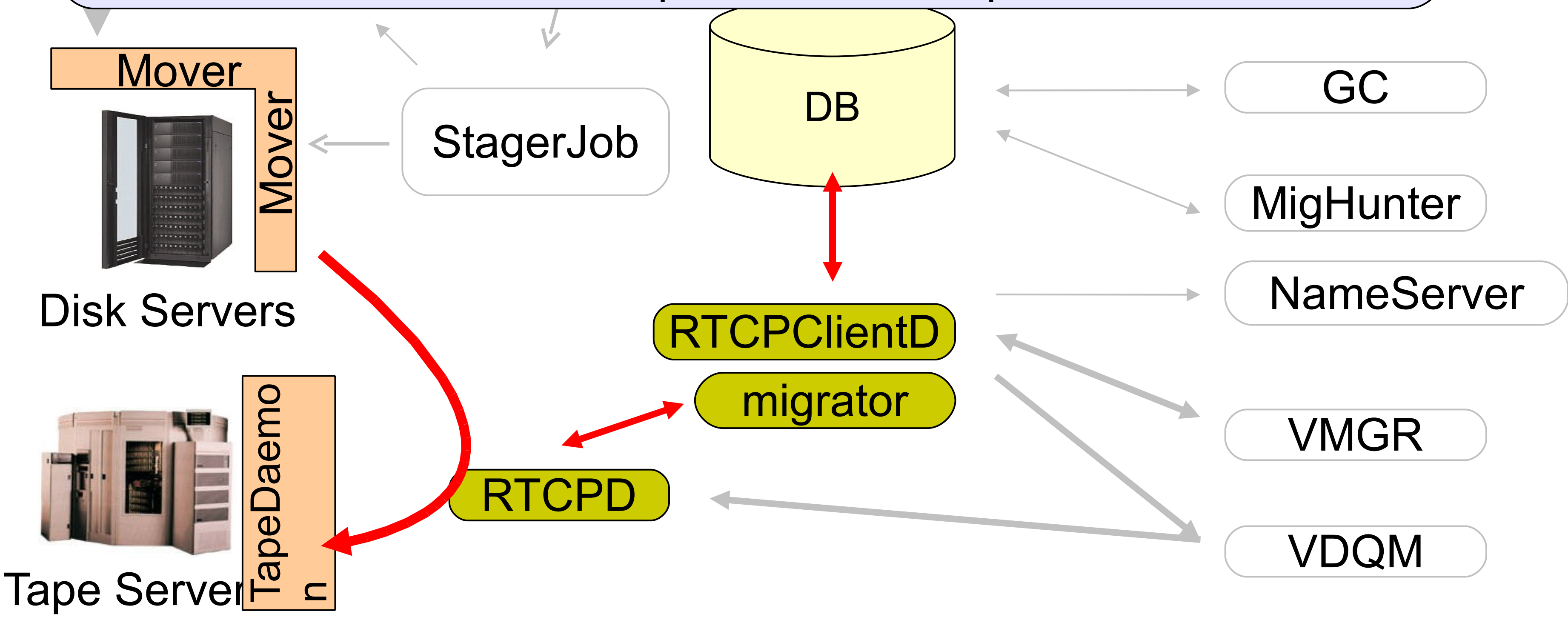
stager_put (4)



- thanks to a MigHunter, the new tapecopy is attached to the streams it can belong to (depending on tapepools, svcclasses, ...)

stager_put (5)

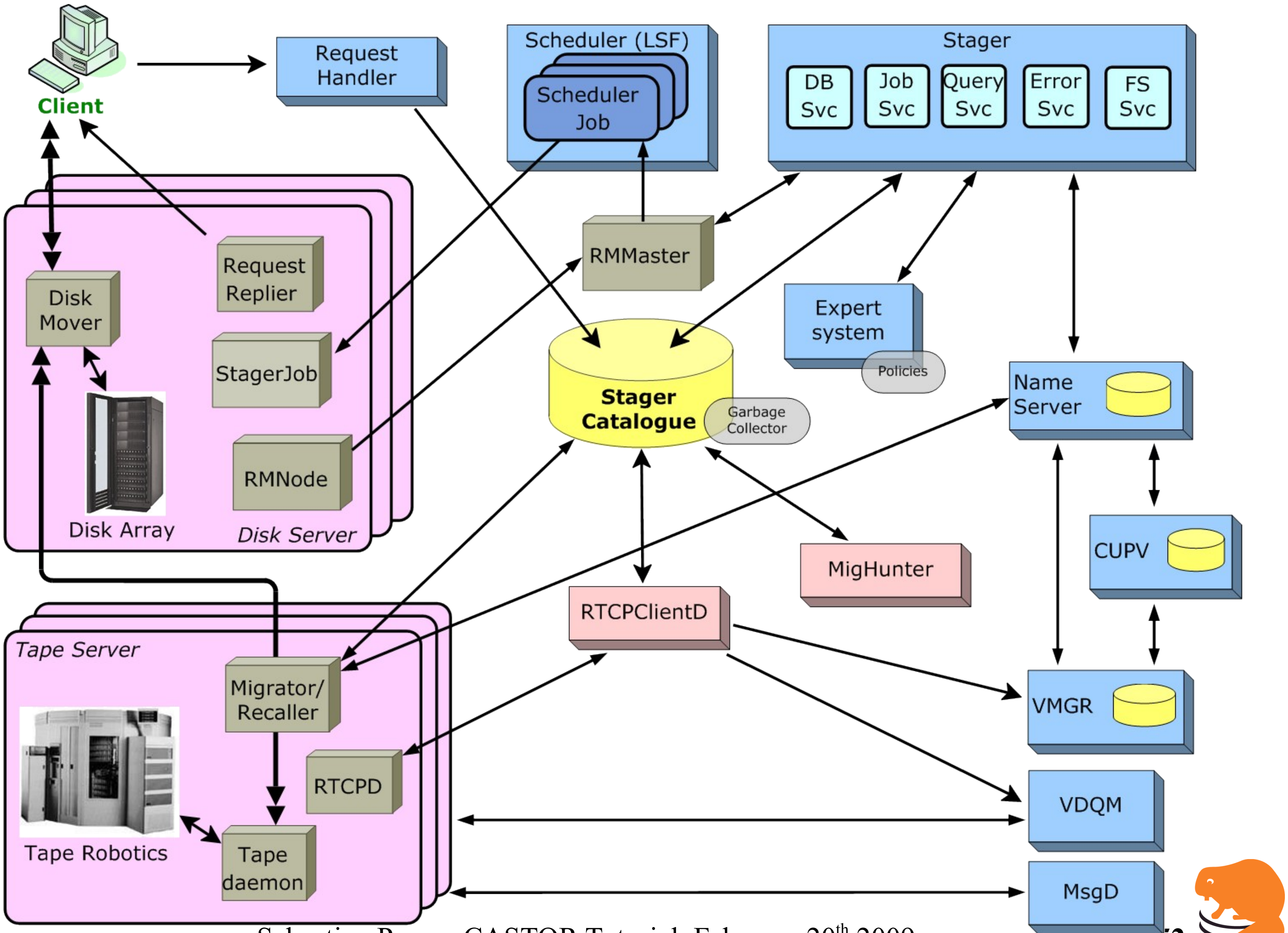
- rtcpclientd will launch a migrator
- this one asks the DB for the next migration candidate
- the DB takes the best candidate in the stream (based on filesystems availability)
- the file is written to tape and the DB updated



Outline

- Detailed view of the architecture
  - Lifecycle of a GET and a PUT request
- Description and status of the components
  - Main daemons
  - Diskserver related
  - Central services
  - Tape related
- Tape migration and recall
  - Workflow details

Detailed picture of CASTOR



Status of all system components

- Request Handler, Stager, JobManager
- the scheduler (LSF) and its plugins
- StagerJob and protocols
- GC DB job, gcDaemon
- RmNode, RmMaster and the shared memory
- Distributed Logging Facility
- Python policies
- NameServer, VDQM, VMGR, CUPV
- MigHunter, recHandler, rtcpclientd, migrator, recaller, rtcpd

Request Handler

- Scope
  - Stores incoming requests into the DB
- Features
  - Very lightweight
  - Allows for request throttling
  - Handles B/W lists
- Maturity
  - Production, stable for years
- Implementation
  - Fully C++
  - Usage of the internal DB API

Stager

- Scope
  - Main daemon for requests processing
- Features
  - Stateless
  - Multi-services implementation by *thread pools*
    - Allows for **independent** services execution, even on different nodes
    - Enhanced scalability
- Maturity
  - Production, stable
  - Few bugs and RFEs, especially around the PL/SQL code
- Implementation
  - Fully C++
  - Usage of the internal DB API

JobManager

- Scope
  - Handling submission of jobs to LSF
  - Babysitting LSF in case of problems
- Features
  - Stateless
  - Automatic cleanups of old/failed jobs
- Maturity
  - Production, stable
- Implementation
  - Fully C++

LSF and its plugins

- Scope
  - Scheduler of I/O access
  - CASTOR plugins select the best FS for each I/O
- Features
  - Redundant, via LSF failover mechanism
  - 2 levels of plugins to make efficient selections
    - First level in C^{++} for rough preselection
    - Second level in python for flexibility
- Maturity
  - Production, stable
- Implementation
  - C++, python

StagerJob

- Scope
  - Executable running on the diskserver and handling one I/O
- Features
  - Supports all protocols (rfio, root, xroot, gridFTP)
    - Via plugin mechanism with defined API
    - 2 levels of support : RawMover and InstrumentedMover
    - Implementing a new protocol is ~200 lines of code
  - Babysits the transfers and handle failures
- Maturity
  - Production, stable
  - Although reimplemented from scratch in 2.1.8
- Implementation
  - Fully C++
  - Plugin mechanism for the different protocols

GC

- Scope
  - Takes decision on which files to delete from the file cache
- Features
  - Policy based, with 3 provided defaults
    - Default, FIFO, LRU
- Maturity
  - Production, stable
- Implementation
  - No daemon, no DB jobs
  - based on weighting diskcopies in response to events
    - e.g. first access, disk to disk copy

gcDaemon

- Scope
  - Deletes files selected by the stager DB in gcWeight order
- Features
  - Stateless daemon implemented as a stager client
- Maturity
  - Production, stable
- Implementation
  - C++
  - Usage of the client API and the internal API
    - proxy “remotized” implementation of the stager
- Note : takes no decision, the GC weight order depends on the implemented GC policy

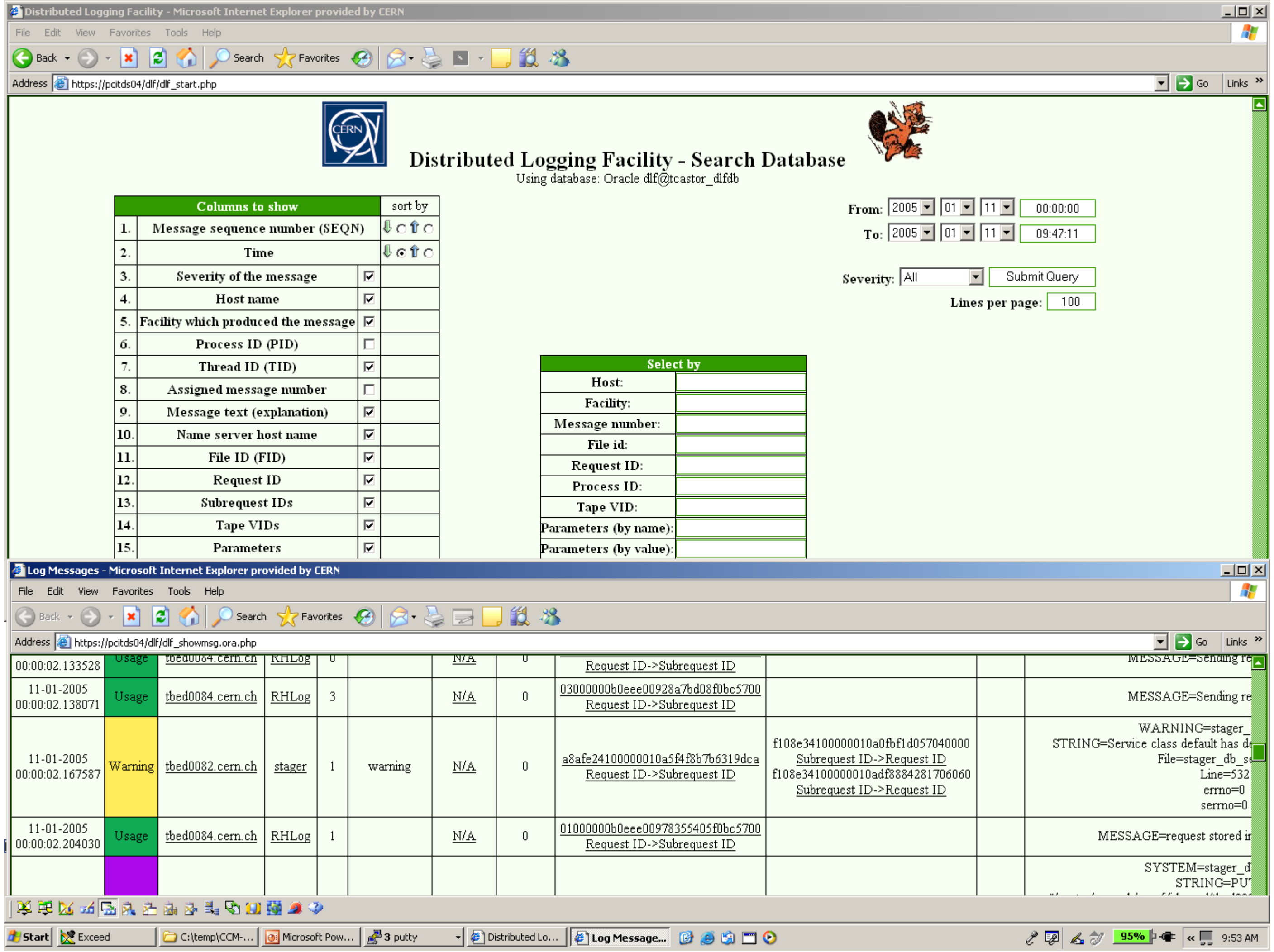
RMMaster & RMNode

- Scope
  - Gather monitoring information from nodes
  - Provide them to other components via shared memory
- Features
  - Must run on LSF master node
  - Redundant via failover mechanism
  - RMMaster gathers data from RMNode
  - RMNode runs on the diskservers and polls `/proc` data
- Maturity
  - Production, stable
- Implementation
  - Fully C^{++}
  - Using shared memory, also accessed by LSF plugins

Distributed Logging Facility

- Scope
  - Central DB-based logging system
- Features
  - A daemon accepts and stores any log entry from any Castor subsystem
  - A PHP-based GUI allows for querying the log
- Maturity
  - Production, stable
- Implementation
  - Fully C, “legacy” DB API

DLF GUI



Distributed Logging Facility - Microsoft Internet Explorer provided by CERN

Address https://pcitds04/dlf/dlf_start.php

Distributed Logging Facility - Search Database

Using database: Oracle dlf@tcastor_dlfdb

	Columns to show		sort by
1.	Message sequence number (SEQN)		⇩ ○ ⇧ ○
2.	Time		⇩ ● ⇧ ○
3.	Severity of the message	☑	
4.	Host name	☑	
5.	Facility which produced the message	☑	
6.	Process ID (PID)	□	
7.	Thread ID (TID)	☑	
8.	Assigned message number	□	
9.	Message text (explanation)	☑	
10.	Name server host name	☑	
11.	File ID (FID)	☑	
12.	Request ID	☑	
13.	Subrequest IDs	☑	
14.	Tape VIDs	☑	
15.	Parameters	☑	

From: 2005 01 11 00:00:00

To: 2005 01 11 09:47:11

Severity: All Submit Query

Lines per page: 100

Select by	
Host:	
Facility:	
Message number:	
File id:	
Request ID:	
Process ID:	
Tape VID:	
Parameters (by name):	
Parameters (by value):	

Log Messages - Microsoft Internet Explorer provided by CERN

Address https://pcitds04/dlf/dlf_showmsg.ora.php

00:00:02.133528	Usage	tbed0084.cern.ch	RHLog	0		N/A	0	Request ID->Subrequest ID			MESSAGE=Sending re
11-01-2005 00:00:02.138071	Usage	tbed0084.cern.ch	RHLog	3		N/A	0	03000000b0eee00928a7bd08f0bc5700 Request ID->Subrequest ID			MESSAGE=Sending re
11-01-2005 00:00:02.167587	Warning	tbed0082.cern.ch	stager	1	warning	N/A	0	a8afe24100000010a5f4f8b7b6319dca Request ID->Subrequest ID	f108e34100000010a0fbf1d057040000 Subrequest ID->Request ID f108e34100000010adf8884281706060 Subrequest ID->Request ID		WARNING=stager_ STRING=Service class default has de File=stager_db_se Line=532 errno=0 serrno=0
11-01-2005 00:00:02.204030	Usage	tbed0084.cern.ch	RHLog	1		N/A	0	01000000b0eee00978355405f0bc5700 Request ID->Subrequest ID			MESSAGE=request stored in
											SYSTEM=stager_d STRING=PUT

Python policies

- Scope
  - Externalize decisions based on policies
- Features
  - Framework for executing python policy scripts
  - Using precompilation to allow for fast execution
  - Used by several components :
    - MigHunter
    - LSF plugin
    - Migrator
    - Recaller

NameServer

- Scope
  - Archive the filesystem-like information for the HSM files
  - Associate tape related information
- Features
  - Stateless daemon, DB backend
- Maturity
  - Production, stable
  - Being improved to improve latencies and response times
- Implementation
  - C
  - ORACLE ProC

Volume and Drive Queue Mgr (VDQM)

- Scope
  - Manage the tape queue and device status
- Features
  - Stateless
  - Supports drive dedication (regexp)
  - Supports request prioritization
- Maturity
  - Production, stable
  - Although was recoded (VDQM2) and new version was deployed only with 2.1.8
- Implementation
  - Fully C++

Volume Manager (VMGR)

- Scope
  - Logical Volume Repository. Inventory of all tapes and their status
- Features
  - Tape pools
    - Grouping of tapes for given activities
    - Counters for total and free space (calculated using compression rates)
- Maturity
  - Production, stable
- Implementation
  - C
  - ORACLE ProC

Castor User Privileges (Cupv)

- Scope
  - Manages administrative authorization rights on other CASTOR modules (nameserver, VMGR)
- Features
  - Flat repository of privileges
  - Supports regular expressions
- Maturity
  - Production, stable
- Implementation
  - C
  - ORACLE ProC

MigHunter/RecHandler

- Scope
  - Mighunter : attaches migration candidates to streams, can hold back streams with no enough data
  - RecHandler : handles recalls priorities, can delay mounts in case only few files would be read
- Features
  - Redundant, SvcClass based
  - Policy based, using python framework
    - Stream policy, migration policy, recall policy
- Maturity
  - Production, stable
  - Still evolving to allow more clever policies
- Implementation
  - C++
  - Usage of python policies

rtcpclientd

- Scope
  - Master daemon controlling tape migration/recall
- Features
  - Not stateless, but restart possible
    - at the expend of tape dismounting
    - But DB inconsistencies are cleaned up
  - Single threaded
- Maturity
  - Production, but has bugs
- Implementation
  - C,
  - Usage of internal DB API
- Will be replaced by the tape gateway

Migrator/recaller

- Scope
  - Controls the tape migration/recall
  - Connects to rtcp daemon on the tapeserver
- Features
  - Forked by the rtcpclientd
  - One migrator/recaller per tape server
- Maturity
  - Production but has bugs
- Implementation
  - C
  - Usage of internal DB API
- Will be replaced by the tape gateway

Tape mover (rtcpd)

- Scope
  - Copy files between tape and disk
- Features
  - Highly multithreaded
    - Overlaid network and tape I/O
    - Large memory buffers allows for copying multiple files in parallel
  - Supports a large number of legacy tape formats…
- Maturity
  - Production, stable
- Implementation
  - C
- Will be extended and then replaces by the tape aggregator

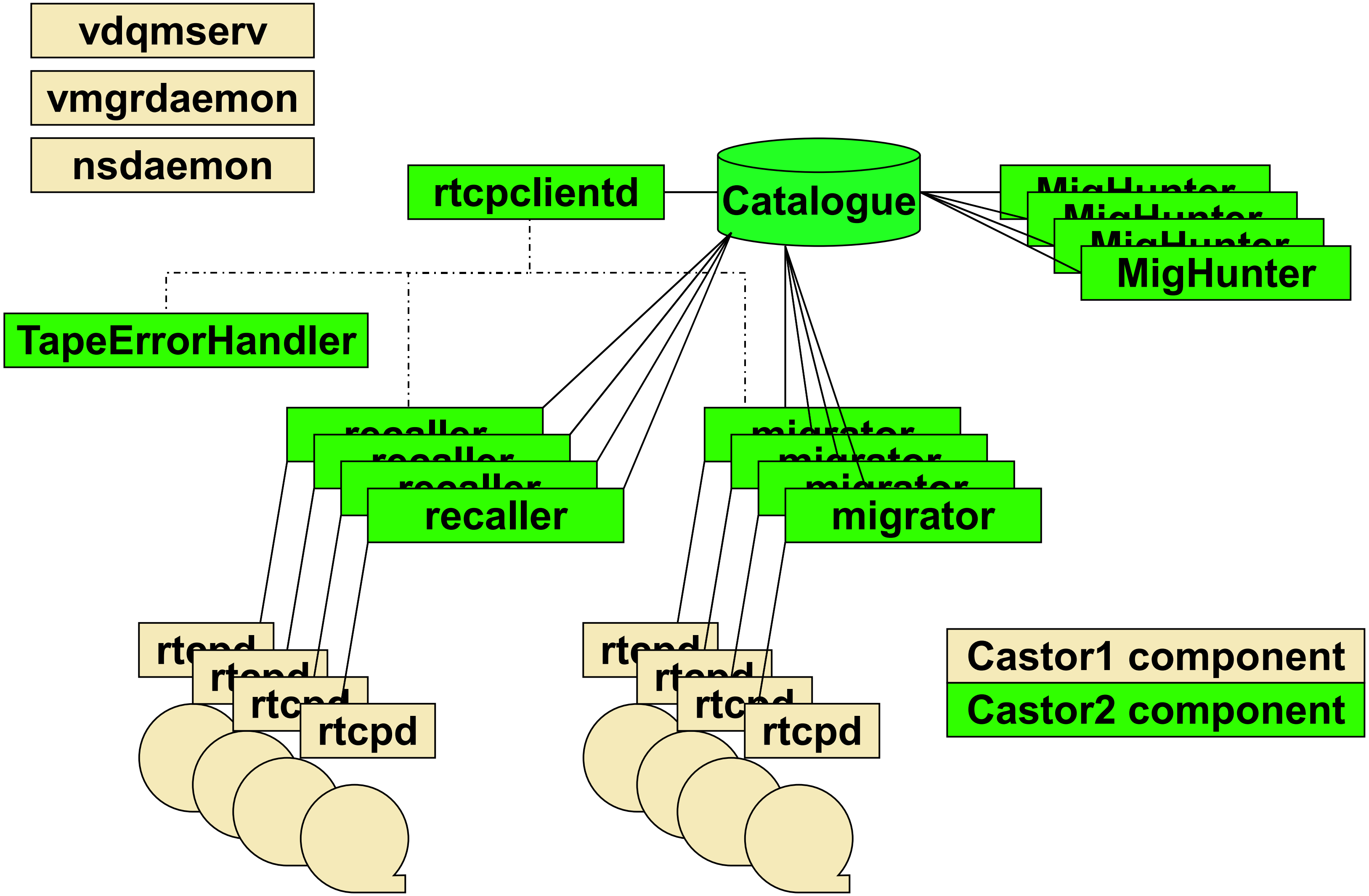
Outline

- Detailed view of the architecture
  - Lifecycle of a GET and a PUT request
- Description and status of the components
  - Main daemons
  - Diskserver related
  - Central services
  - Tape related
- Tape migration and recall
  - Workflow details

Tape Migration and Recall

- “rtcpclientd” is the main component dealing with all interaction to the CASTOR tape archive
  - For each running tape recall it forks a ‘recaller’ child process per tape
  - For each running tape migration it forks a ‘migrator’ child process per tape
- Migration streams are created and populated by the “MigHunter” component
- A TapeErrorHandler process is forked by the rtcpclientd daemon whenever a recaller or migrator child process exits with error status.

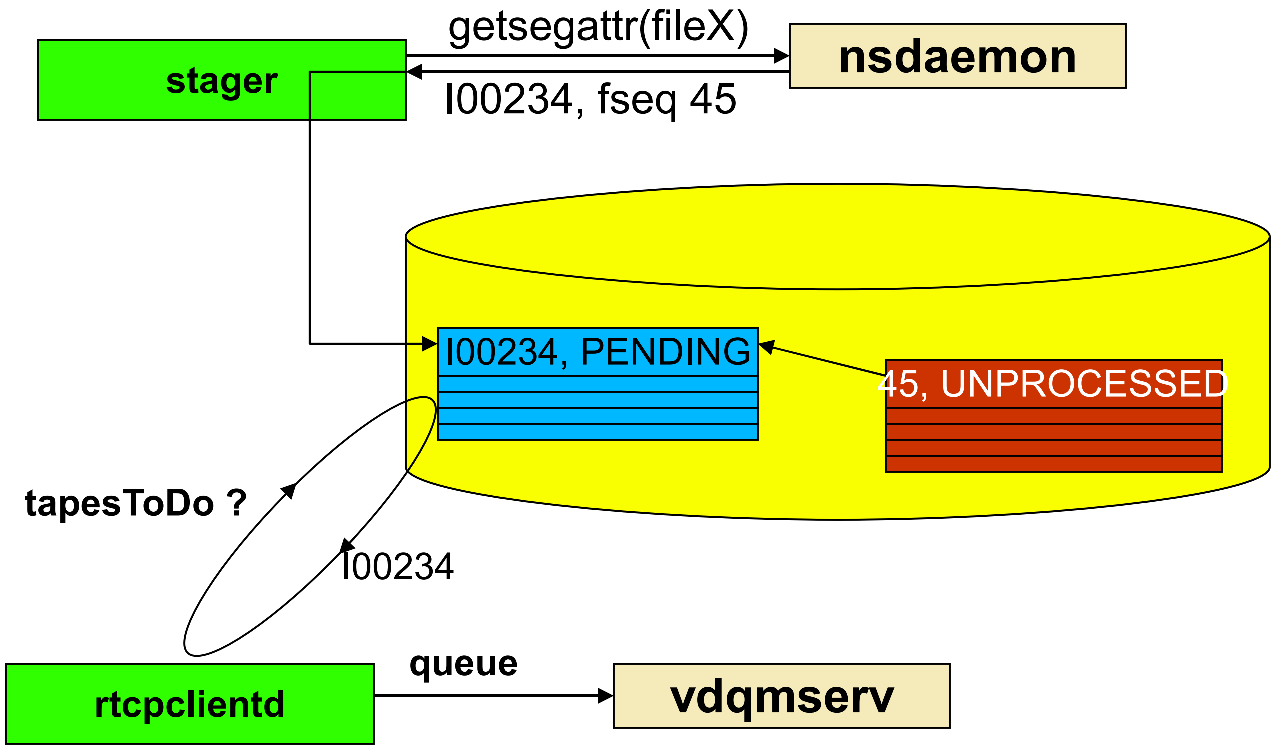
Tape migration/recall components



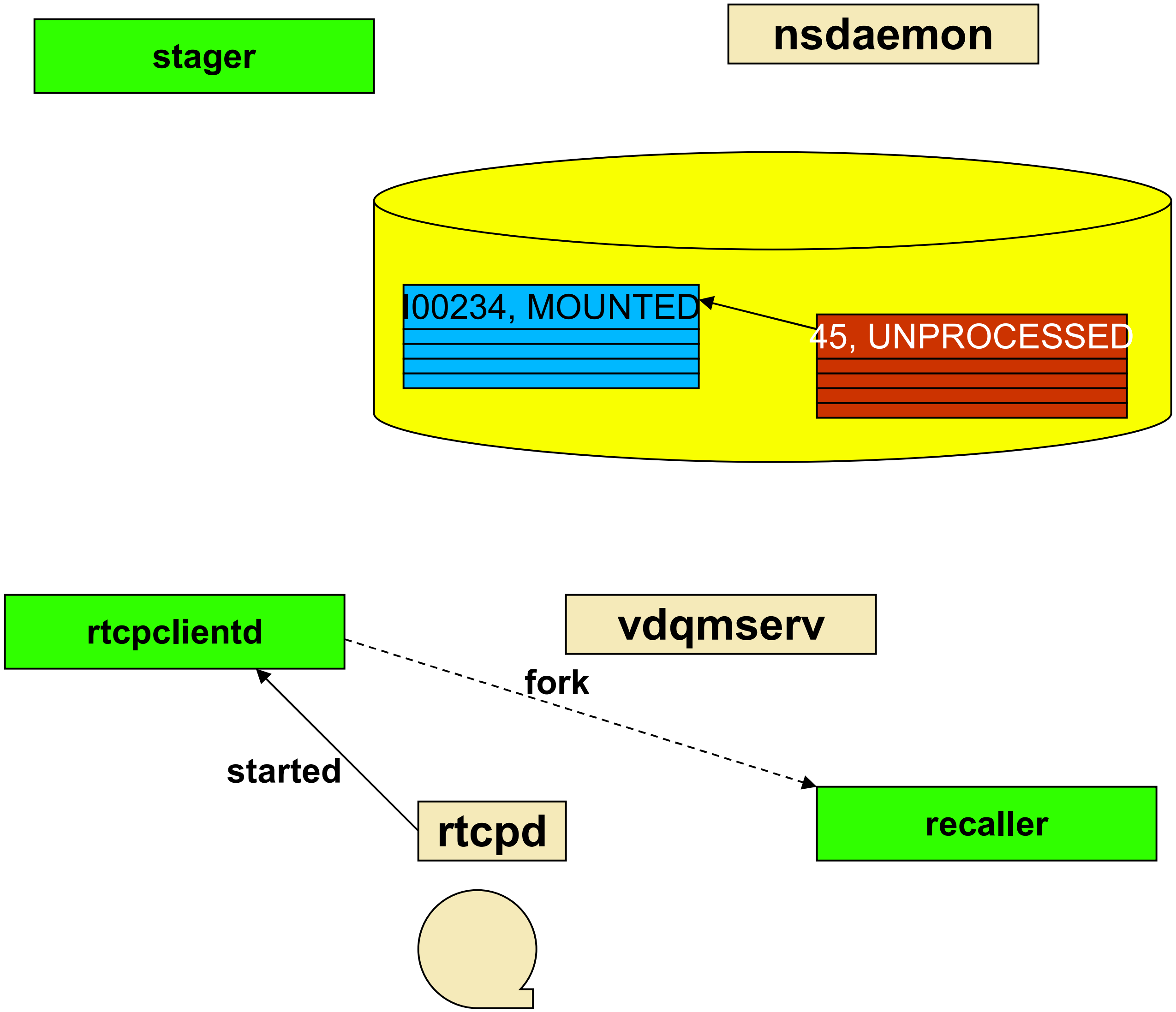
Tape recall (1)

- Triggered by disk cache misses in the stager
  - Stager calls the name server to retrieve the tape segment information (VID, fseq, blockid)
  - Stager inserts a Tape and Segment in the DB
    - With status depending on the use of recall policies
- RecHandler goes through recall candidates
  - Enables waiting recalls according to policy
  - Handles recall priorities and inform VDQM
- Rtcpclientd checks the DB for tapes to be recalled
  - Submits the tape request to VDQM (tape queue)
  - When mover (rtcpd) starts it connects back to the rtcpclientd, which then forks a recaller process for servicing the tape recall

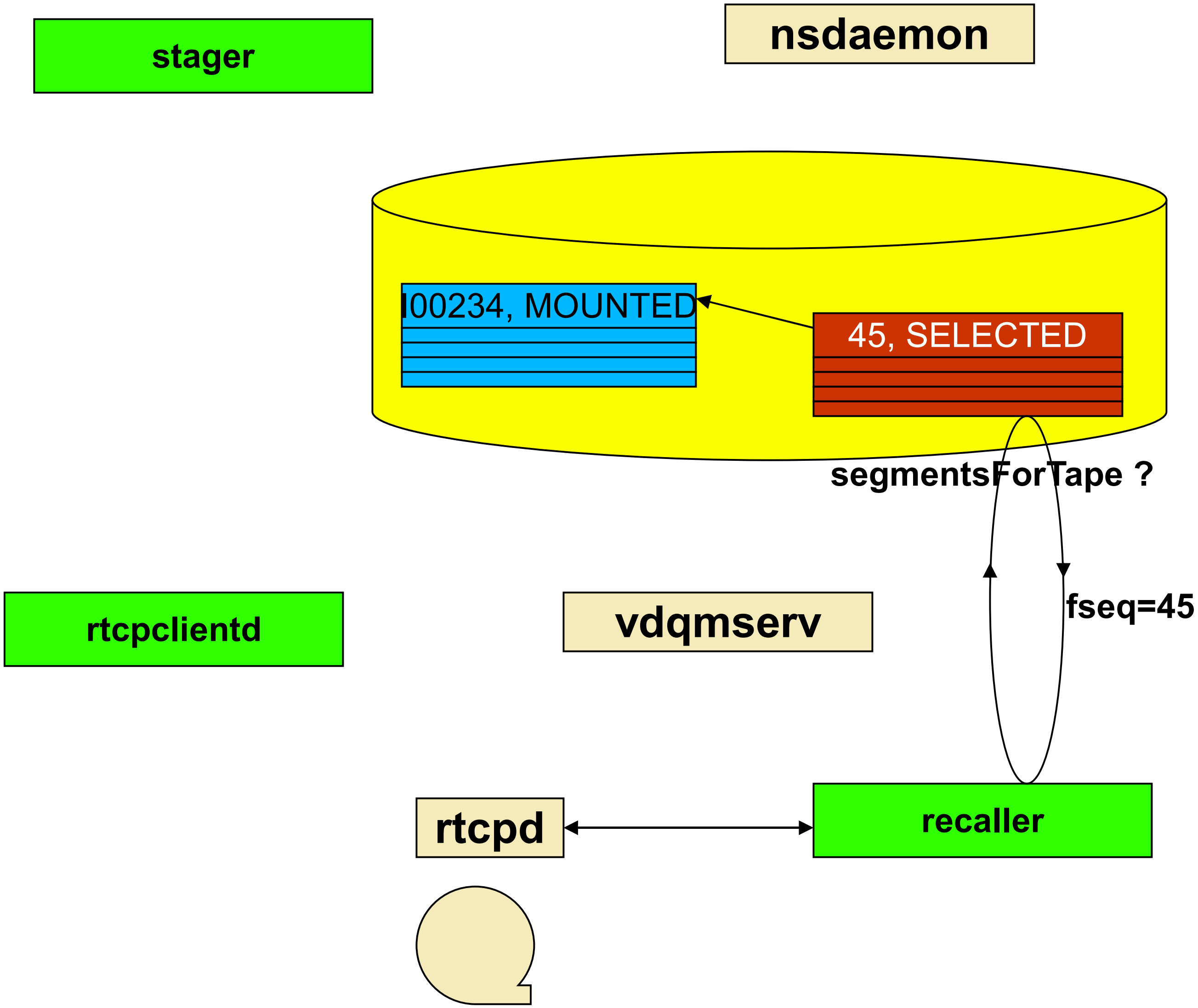
Tape recall flow



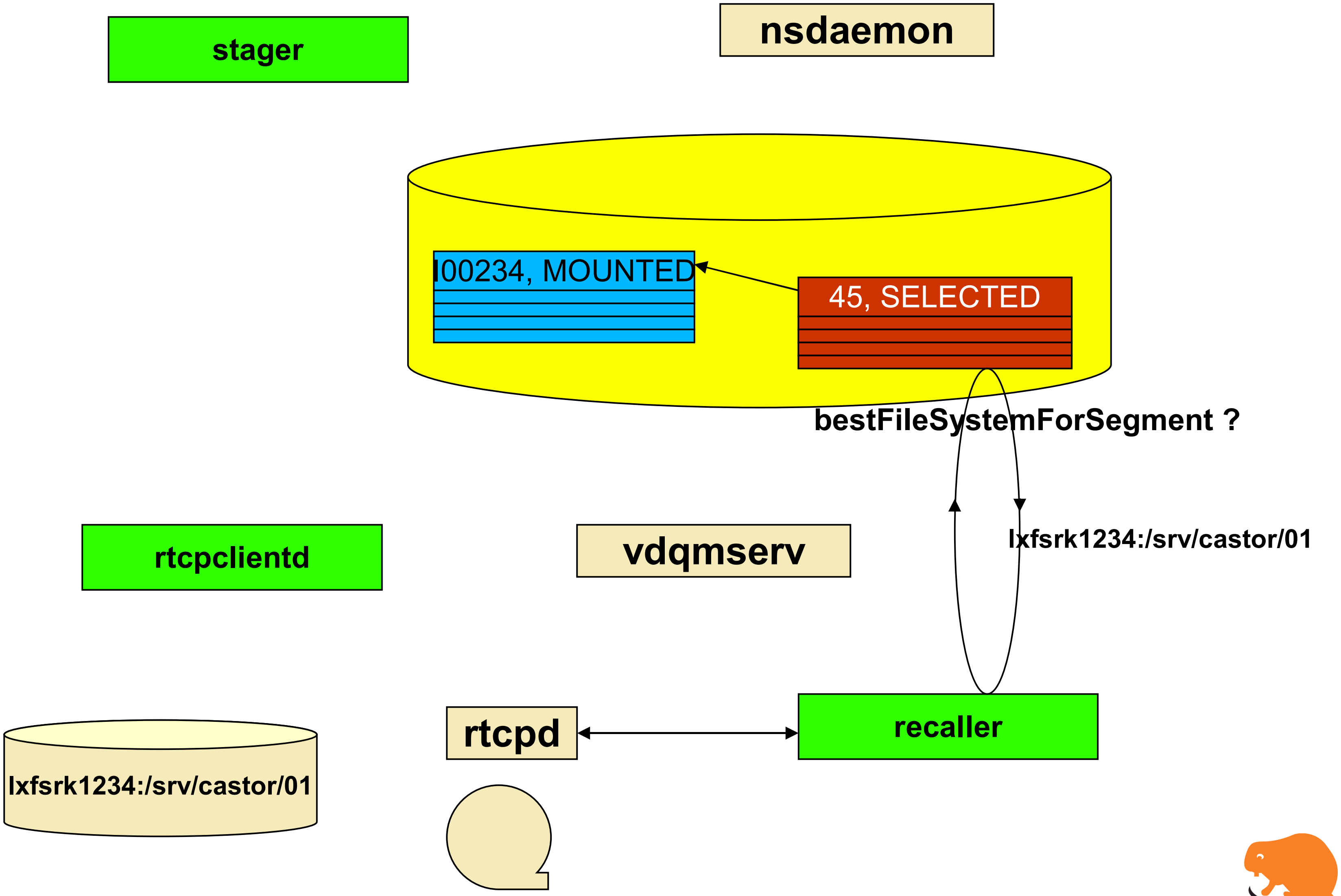
Tape recall flow



Tape recall flow



Tape recall flow


stager
nsdaemon
I00234, MOUNTED
45, SELECTED
bestFileSystemForSegment ?
lxfsrk1234:/srv/castor/01
rtcpclientd
vdqmserv
rtcpd
recaller
lxfsrk1234:/srv/castor/01

Tape recall (2)

- The recaller attempts to optimize the use of tape and disk resources
  - Tape files are sorted
    - Current in fseq order. Work needed to find more optimal sorting taking into account the serpentine track layout on media
  - Requests for new files on same tape are dynamically added to running request
  - Target file system is decided given the current load picture when the tape file is positioned

Tape migration

CERN IT Department

- Similar to tape recall but
  - Triggered by policy rather than waiting requests
- Migration candidates are attached to ‘streams’
  - A migration ‘Stream’ is a container of migration candidates
  - Each Stream is associated with 0 or 1 tapes:
    - 0 tape: stream not active (e.g. not yet picked up by rtcpclientd, or VMGR tape pool is full)
    - 1 tape: stream is running (tape write request is running) or waiting for tape mount
  - A Stream can survive many tapes (but only one at a time)
  - A TapeCopy can be linked to many Streams
    - When a TapeCopy is selected by one of the Streams, its status is atomically updated preventing it from being selected by another Stream
- The MigHunter process is responsible for attaching the migration candidates to the streams
  - Migration and stream policies can be used for fine-grained control over this process

Example migration policy

```
def smallFilesMigrationPolicy
  (tapePool, castorfilename, copynb, fileId,
   fileSize, fileMode, uid, gid, aTime,
   mTime, cTime,fileClass):

  if ((fileSize <= 30000000) and
      (tapePool == smallfilesTP)):
    return 1
  elif ((fileSize > 30000000) and
        (tapePool != smallfilesTP)):
    return 1
  else:
    return 0
```

Example stream policy

```
def defaultStreamPolicy
    (runningStream, numFiles, dataVolume,
     maxNumStreams,status) :

  wantedStreams = int(numFiles / 1000)
  wantedStreams = wantedStreams%(maxNumStreams+1)
  if wantedStreams>runningStream:
     return 1
  else:
     return 0
```

That's it for now