

CESNET Technical Report 21/2009

Transition to Inter-Cluster Scheduling Architecture in MetaCentrum

MIROSLAV RUDA, ŠIMON TÓTH

Received 15.12.2009

Abstract

For last ten years, scheduling of computational jobs across MetaCentrum (Czech national grid) was managed by one, central PBSPro installation. Reason for this decision was the possibility to schedule jobs between different clusters (spread across whole Czech Republic), with full understanding of complete situation of all clusters, with shared fair-share policy for users and with better support for large jobs, running across different clusters. Development effort was concentrated on improving stability of this setup (especially in case of instability of the national network connecting different clusters) and support for advanced scheduling methods and virtualization. Yet, with the growing number of clusters and processor, this setup is becoming problematic and may become single point of failure and scalability bottleneck. In this paper we study possibility of change MetaCentrum scheduling system to the system of less depended clusters, each maintained by separate server and scheduler, but still fulfilling original requirements on central accounting of jobs, fair share of computational resources across complete MetaCentrum and possibility to schedule large jobs or virtual clusters across such infrastructure. Because several of the reasons to choose PBSPro usage are also invalid in such setup (PBSPro was chosen for its better stability in such large setup and a better scheduling system supporting large number of jobs), we are also evaluating the possibility to switch scheduling system from PBSPro to open-source Torque system. Main features of PBSPro, used by MetaCentrum, are enlisted, together with discussion of state of such features in Torque, possible replacements and required development of missing features.

Keywords: National Grid, Torque, scheduling

1 Introduction

Current MetaCentrum¹ job scheduling system is based on a central installation of PBSPro², managing all jobs in different clusters across whole Czech Republic. Managed clusters are spread over several cities (Brno, Prague, Plzeň, Č. Budějovice) and even more organizations (CESNET and Charles University in Prague, University of West Bohemia in Plzeň, University of South Bohemia in České Budějovice, Masaryk University, Technical University, and Mendel University of Agriculture and Forestry in Brno).

This setup allows the scheduler to plan jobs between different physical clusters with the knowledge of complete grid state, complete fair-share information and support for big jobs running across multiple physical clusters. It enables high quality scheduling through the whole grid, provides a simple load balancing between clusters, etc.

¹ <https://meta.cesnet.cz>

² <http://www.pbspro.com>

These advantages were until recently outweighing the disadvantages, but with steady growth of computational capacity, number of clusters (new clusters from universities in different cities have joined MetaCentrum in last years) and the number of jobs, we are slowly reaching limits of the central solution in several aspects:

- Stability – we have significantly improved stability of PBS on such geographically large-scale installation, managed to boost the stability of the server in case of instability of national network connecting different clusters, but network problems in one site (metropolitan network problems, problems with local routers/switches) will always have negative impact on the whole system. With growing number of sites managed by less trained administrators and with clusters connected to national backbone via non-dedicated network, such outages are to be expected and will become regular.
- Scalability – with the growing number of jobs and nodes managed by central server, we are loosing the ability of the server to monitor such amount of nodes and jobs reliably. Similarly, scheduler is reaching its limits too. With large amount of jobs submitted into queues at the same time we are already experiencing unexpected hiccups and general slowdown of scheduling system. The situation will be unbearable with the expected growth of computational capacity and especially with the expected increase of small jobs ratio (parametric studies).
- Single-point-of-failure bottleneck of such solution is becoming more visible, network outage between the clusters disallows local users in one site to use even local resources. HA feature from PBSPro is not very usable, because it still relies on shared filesystem between primary and secondary copy of PBS servers. Even when using High Availability support, we still can't maintain functionality in case of inter-city network connectivity drops, or network breakdowns in Brno (the location of the main PBS server). This problem is amplified by the fact that we not only lose the ability to run inter-cluster jobs, or load-balancing, but we lose the ability to run any jobs (including local) in the affected area. Network breakdowns also cause cascade effects on the whole grid performance. Due to the vast geographic area covered by the grid, the server has to be configured with higher timeouts. Each downtime causes the server to wait for each non-reachable nodes.
- Pricing policy of PBSPro, where license fee is based on number of CPUs. Growing number of CPUs imply not only higher price for such solution, but also generates additional expenses when adding new cluster to national grid.

As a result, a new scheduling architecture is proposed, which relies on higher autonomy of clusters. It is based on a peer to peer network of semi-independent schedulers for each site or even cluster. Individual schedulers maintain their assigned clusters—allowing e.g. to submit jobs locally even if the external connectivity is lost—while cooperating with its peers to support features that mimics the centralized planning. Namely, the system still supports central accounting of jobs, fair share of computational resources across all sites of the MetaCentrum and scheduling jobs across all resources. Other proposed features are: support for inclusion of third-party middleware (based for example on SGE/PBS/LSF installation), support for large jobs running across several clusters and one entry point for users

independently on their current location (with backup solutions in case of network outage, preferably with the same functionality on all instances). As MetaCentrum provides also virtualized resources (moving towards cloud provisioning [7]), the scheduling system is being integrated with the Magrathea system to support scheduling of virtual clusters.

1.1 Proposed Solution

Details of the proposed solution are discussed in the following sections, including a short Torque introduction, analysis of its peer to peer capabilities, presentation of already implemented features, analysis of planned features and list of future challenges.

Current proposal is based on separate Torque³ installation in each organization (with reasonable autonomy in case of network problems), local gateway (defined in next section) providing access to the whole grid on each site and implementation of scheduler modifications, supporting scheduling over the whole MetaCentrum. In the following sections, main features of these components will be defined.

We are considering the Torque batch system as the alternative, and not some completely different batch system like SGE⁴ or Slurm [8], for several reasons. Both Torque and PBS-Pro have common code-base roots in OpenPBS [14]. This allows us to port custom changes we implemented into PBS-Pro relatively easily. Common code base also grants compatibility for users, Torque offers practically identical user interface. Torque is also the most commonly used grid backend that is supported by almost all grid interfaces.

The usual approach to distributed scheduling using small separated clusters is the hierarchic model. Each Torque installation has its own separate grid gateway like Globus [1] or gLite [15], with job management systems like Condor [3] or Gridway [12] operating on top of these gateways. We still need to be compatible with these systems (we have to provide our gateways), however we would like to maintain better scheduling features between clusters for the national grid. Hierarchic systems do not possess enough information for real planning and usually only assign jobs into clusters depending on their current load. These systems often offer low possibilities of cluster rebalancing upon clusters state change (see job pilots initiative in EGEE [11]) and provide very low support for applications using resources from multiple clusters at once. Due to these limitations, we are evaluating the possibility of inter-cluster scheduling using the Torque batch system itself.

1.2 Grid Interface

Proposed architecture must be compatible with standard global grid implementations, especially gLite interface (CREAM [10], LCG CE⁵) to EGEE/EGI grid and Globus interface (used within smaller grid installations and very often used for short-time setups or demonstrations). However, instead of providing such in-

³ <http://www.clusterresources.com/products/torque-resource-manager.php>

⁴ <http://gridengine.sunsource.net/>

⁵ <https://twiki.cern.ch/twiki/bin/view/EGEE/LcgCE>

terface for each cluster (and relaying on meta-schedulers like Condor or Gridway for scheduling across clusters), we study architecture, where such gateways may be installed on several clusters and provide access to the whole grid using our peer-to-peer scheduling between individual clusters.

Torque installation managing one cluster is the most favored and supported option in standard grid middleware, modified scheduler should be completely invisible to such gateways, which are usually using simple interface of command-line tool like `qsub`, `qstat`, `qdel` or Distributed Resource Management Application API (DRMAA) [9]. Because we will not change this API neither Torque server API, compatibility should be preserved.

Compatibility with clusters managed by other batch systems, such as LSF [4], Slurm [8] or Torque+Maui [2], will be also studied. Torque-based systems, with external schedulers like Maui or Moab⁶ could be supported via specialized routing queues supported by Torque. For other systems, possibility of deploying some form of gateway will be studied, either following standard Globus GRAM interface [13] implementation, or using our proposed gateway service. In partially independent work, possibility of better support for HTC (high-throughput systems) systems, like Condor or some pilot-job based systems, will be studied, too.

2 Technical Background

2.1 Batch System Anatomy

PBS based batch systems are composed of three basic components.

The *Server*, which is responsible for coordination and information storage and also serves as a mediator in almost all operations. *Scheduler*, responsible for decision making about which jobs will be run, when they will be run and where. The last component is a set of computational nodes (in the PBS-Pro/Torque terminology called *MOMs*). Nodes are used to execute jobs. Different nodes can provide varying specific features like number of processor, amount of memory and special hardware like specialized graphic cards.

Managing jobs is the main responsibility of the batch system. Although jobs can have very different properties (like time, they need to run or various resource requirements) we will usually speak about a *Job* as a single unit to describe the batch system.

Each job enters the system by being submitted to one of the server queues (each server can have an arbitrary amount of queues with different properties). Job will remain in this queue until it is executed on one of the nodes, or until it is transferred to another queue.

⁶ <http://www.clusterresources.com/products/moab-cluster-suite.php>

2.2 Current vs. Proposed Architecture

Currently used architecture contains always only one server, one scheduler and certain number of nodes. When the scheduler is the only one talking to the server, it can work with the knowledge that the server's state is consistent with its assumptions.

Most common setup involves a Torque cluster with one server and one free or commercial scheduler (Moab or Maui). In this case, the server only serves as a job storage and all logic (including resource management) is done on the scheduler side.

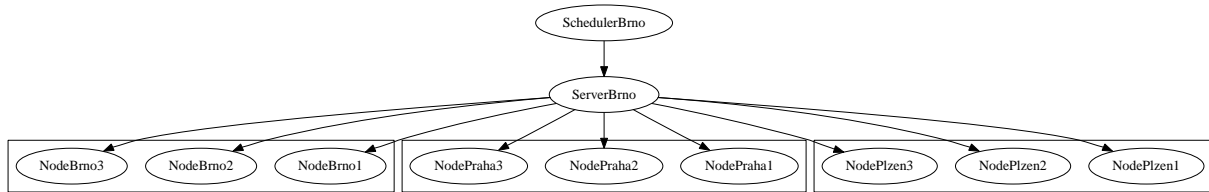


Figure 1. Currently used architecture

When switching to a distributed architecture, we no longer have a consistent state. Each scheduler is talking to each server and this, due to the servers asynchronous nature, can lead to scheduler request interleaving (atomicity on the single-command level is still preserved).

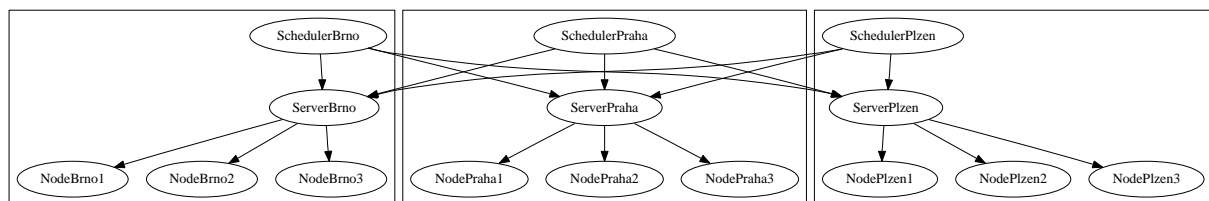


Figure 2. New proposed architecture

2.3 Torque Setup

We want one installation of Torque in each organization, at least one for each site, but there can be even more than one Torque server if organization is spread across several buildings, up-to one Torque per physical cluster.

- One torque server manages only its own cluster (monitoring of nodes and jobs in the cluster). This architecture solves the basic problems of one-server setup. We have reasonable size of each cluster, one cluster downtime (or network outage between clusters) will not influence other clusters.
- Torque server has special routing queue for new jobs and standard queues for jobs designated to local cluster.
- Modified Torque scheduler (not Maui/Moab scheduler, our solution will be based on the simple internal FIFO scheduler, which will be extended with needed features) will be installed together with each Torque server (but mapping doesn't have to be necessary 1:1, see next sections).

- Scheduler’s task is to schedule jobs from corresponding routing queue’s on both local and remote clusters and schedule local jobs on the local cluster. It will read information both from local server (jobs in routing queue, local jobs, local nodes) and from all other Torque servers (mostly information about nodes, but potentially about other jobs too). Because each scheduler talks to a reasonable amount of servers, we are maintaining a reasonable scalability.
- Completely independent scheduler, providing swapping jobs from local queues to other clusters can be also implemented.
- Torque supports job moving between servers, we might need to implement “shadow” jobs that will not leave Torque even after moving the jobs, so we won’t confuse monitoring tools and gateways.

2.4 Gateway

Gateway will be installed together with each Torque server, it will accept job from the user, put it into routing queue on local Torque installation. Gateway will also provide user interface for job monitoring, including jobs submitted or moved to remote clusters.

Each gateway will

- support familiar PBS API – job definitions, resource description etc.;
- unify the environment by hiding specific configuration requirements of individual batch system installations;
- allow VO (virtual organization) specific profiles, providing default parameters, settings and views tuned for specific VO;
- provide monitoring data for whole MetaCentrum;
- list all user jobs, directly when using local Torque, using some form of cache or Logging and Bookkeeping service when accessing remote servers. We will study usability of general LB+Provenance installation on top of Torque for this functionality.
- be implemented as web interface, with command-line interface compatible with standard PBS (qsub, qstat);
- provide information about virtual clusters, can list them and submit new jobs into them (if there is a batch system inside), maybe even create a new gateway for the virtual cluster.

2.5 Transition Roadmap

Transition roadmap is being laid down. Several main tasks were already defined, some of them are more deeply discussed in this paper:

- Verify Torque stability in site installation, transfer our stability patches from PBSPro.
- Port our changes from PBSPro – Kerberos support, IPv6 support, scheduling improvements, Magrathea support. Discussed in Section 7 in this paper.
- Implement missing PBSPro features, identified in Section 7 in this paper.
- Propose and implement required modification in the scheduler (support for multiple servers), needed changes in the scheduler logic. First implementation proposal is discussed in Section 4.

- Re-implement features lost by distributed architecture – central accounting, fair-share of resources between various virtual organizations across the whole MetaCentrum. See Section 4 for more in-depth discussion.
- Modify Torque MOM to support multiple servers (jobs running over multiple physical clusters), or support for node swapping between servers, required for jobs running across several clusters, as described in Section 5.
- Implement gateway for users and in later phase, gateway for different batch systems.

3 Torque Components with Impact on M:N Architecture

This section provides an overview of important Torque components that will influence our M:N architecture transition.

3.1 (FIFO) Scheduler

FIFO scheduler is a sample implementation of a scheduler in the C language.

The communication between scheduler and server is initiated by the server which sends a wakeup command containing information about the event that caused the wakeup (`schedule()`). The supplied FIFO scheduler does not differentiate between single commands and only performs 4 different actions for the following groups of commands.

Commands `SCH_ERROR`, `SCH_SCHEDULE_NULL`, `SCH_RULESET` and `SCH_SCHEDULE_RECYC` are completely ignored. After receiving the `SCH_QUIT` command, the scheduler stores the current fair-share information (if any) and exits the scheduler daemon. After receiving the `SCH_CONFIGURE` the scheduler first stores the current fair-share information and then reloads and reinitializes settings. After receiving `SCH_SCHEDULE_NEW`, `SCH_SCHEDULE_TERM`, `SCH_SCHEDULE_FIRST`, `SCH_SCHEDULE_CMD` or `SCH_SCHEDULE_TIME` a full scheduling cycle is run.

After the new connection from server is accepted and the command received, scheduler overtakes the initiative. Scheduler initiates requests to read current server state, information about current nodes and their state, list of queues and their content (`query_server()`). These commands are blocking on the scheduler side and are sent using the same connection, that was established by the server during the wakeup call. While processing information about nodes, the scheduler can also optionally contact each of the nodes (`talk_with_mom()`) to receive further information about their current state like node architecture, amount of physical memory, number of CPUs, etc. The proposed new architecture requires this information to be relayed by the server, therefore we won't be using this feature.

After the current state of the cluster is received, scheduler continues by updating information about fair-share (by comparing the list of currently running jobs with a list of jobs the scheduler expects to be running).

The main scheduling cycle is composed of a simple search for jobs (`next_job()`) that can be run on the cluster with the current state. This search is done using priorities set in the configuration (fair-share, queue priorities, round-robin). The scheduler implicitly prefers running small jobs (with small resource requirements),

because the scheduling cycle always tries to run all jobs (and small jobs, even with small priority are likely to fit into gaps left by big jobs).

If the scheduler finds a job suitable to be run on the cluster (`find_best_node()`) it sends a run command to the server (`run_update_job()`). In the default implementation, if the server is not using timeshared nodes, the scheduler does not specify on which node should the job run and leaves this decision to the server.

The FIFO scheduler works in a loop: *receive command* → *read info* → *foreach_{job}* (*schedule* → *send commands*) → *receive command*.

Due to this behavior it can work with multiple servers (with the exception of fair-share information which won't be calculated correctly when multiple servers are used).

3.1.1 Fair-share

Fair-share is represented in the scheduler using a structure (`group_info`) that contains information about accumulated workload generated by groups and single users and also containing target ratios of expected groups and users workload. This structure is tree-like and can define ratios both system-wide and group-wide (group A can have 50 % of total system load, and its sub-groups C and D can have 40 % and 60 % of group A load, resulting in 20 % and 30 % in total system load).

Fair-share is updated using the information stored in a separate structure (`last_running`). This structure contains list of jobs that are expected to be running (were running on last wakeup). This list is updated on the end of each scheduling cycle.

Information about last running jobs is compared with the list of currently running jobs (this information is downloaded from the server). If a job is found both in the data structure and the current running list the owner of the job (including all his groups) receives additional workload corresponding to the product of cputime and number of CPUs.

When the scheduler is not running, the fair-share information is not updated. More specifically, fair-share information will be lost for the time the scheduler was not running (plus the time from last fair-share synchronization if the scheduler crashed), but only for jobs that terminated during the downtime. If the job is still running when the scheduler is brought back up, fair-share information will be update correctly.

Old fair-share data should be less relevant than current data. This is achieved using periodic decay (`decay_fairshare_tree()`), which is run on the whole group-user tree, halving all values. The period can be configured in the scheduler settings.

If fair-share usage is turned on, job priority is determined using the fair-share priority (also following other scheduler settings – round-robin, global queue...). Fair-share priority is calculated as ratio between currently accumulated and expected system-wide load.

3.2 Nodespec

Nodespec represents a way how users can impose specific requirements upon nodes that will be selected to run the job. For example a user might request one node with a printer and three nodes with a graphic card.

Nodespec is specified during submit using the `qsub -l` parameter⁷. Basic format for nodespec is:

```
mom_req+mom_req+...#mom_req#shared,
```

where `mom_req` is in the format

```
amount:property1:property2=amount_property2:property3.
```

Requirements specified after hash are global and will be applied to all previously specified nodes (we can for example request all nodes to be 64bit). The second use for hash mark is specifying whether we request shared or exclusive access to nodes.

If we don't require the whole node just for us (for example, our 4 cpu job will receive a 16 cpu node, the remaining 12 cpus can be assigned to other jobs), then we can specify the `#shared` property. Default value is exclusive access which will only allow one job per node.

3.2.1 Nodespec Example

```
3:ppn=2:amd64+1:ppn=1:printer#san#shared
```

We request 3 nodes, 2 processes (one cpu each) on each, 64bit and one node, 1 process, with a printer. All nodes can be shared and must have SAN access. All used properties (64bit, san, printer,...) must be supported by the server.

3.3 Resource Reservation

Torque supports reservations on node level. The reservation function is expecting a nodespec and therefore can allocate both a specific node (each node has a property hostname) as well as any node fitting specific requirements.

Reservations are only recorded in the server structures therefore if we allow node sharing (between servers) they have no practical meaning.

Torque also supports partial reservations. If a request for reservation cannot be fully satisfied, nodes are still marked as reserved and its the reservation owners job to either keep these nodes, or free them.

3.4 Authorization

The general authorization scheme is based on lists of allowed clients. All three Torque components differentiate between connections from non-privileged or privileged ports. Non-privileged connections are either completely refused or handled specifically.

⁷ `qsub -l nodes=3:grcard+1:printer`

Connections from privileged ports receive maximum rights (with the exception of the server, which also works with ACL lists).

In this section we will use the word client for the initiator of the connection (in the context of the server-client architecture). Word server will be used in both “Torque server” and the server-client architecture context.

3.4.1 Local

If the communication is local and is handled through Unix sockets, authorization is provided using information in these sockets.

3.4.2 Scheduler

Scheduler works with a list of clients that are allowed to talk with it and does handle non-privileged connection specifically. Connections coming from privileged ports are compared to the list of allowed clients (`okclients`), which is specified in the configuration file in the format `$clienthost hostname`. Connections coming from non-privileged ports are compared to the wildcard client list (list of clients that can provide RPP services on non-privileged ports) (`maskclient`), which are specified in the format `$restricted hostname`.

The current FIFO scheduler does not initialize any new connections (with the exception of reading detailed information from nodes), all communication is handled through the connection initiated by the server during the wakeup call (`server_command`).

3.4.3 Server

Server stores authorization information separately for each connection. If ACL is turned off, the server accepts all connections coming from privileged ports. Connections from non-privileged ports have to be explicitly authorized using the `PBS_BATCH_AuthenUser` sent by `pbs_iff`.

Communication with node is only allowed if the node is in the server node list.

3.4.4 Nodes

Nodes work with allowed clients list. This list is specified in the configuration file (`$pbsserver hostname`) (`$pbsclient hostname`), but also using `IS_CLUSTER_ADDR` command sent by the server. These commands contain the list of nodes for the server.

Node receives `RM_PROTOCOL` and `IM_PROTOCOL` messages from both server and clients.

`IS_PROTOCOL` messages are only received from servers.

`IS_CLUSTER_ADDR` message is a tool to update nodes current sister list.

3.4.5 User Authorization

To verify if the user has rights to run a job on a specific server the `Libsite` (`site_check_user_map()`) library is used. This library internally uses `ruserok`.

Most parts of the authorization in this library are only skeletons, ready to be modified for specific needs.

When transferring job outputs, Torque expects a correctly configured scp, that will transfer files between machines without asking for password.

4 World Scheduling

This section describes the current state of our Torque extensions towards the M:N architecture support.

Scheduler directly supports sequence scheduling of multiple servers. Each scheduling cycle is a separated block, after which all temporary information is discarded (with the exception of fair-share).

First step to modify the scheduler to work with M:N architecture was therefore implementation of new data structures that will hold persistent information about the world.

4.1 Data Structures

Information about the whole cluster (or at least the part that is contacting one specific scheduler) is kept in one data structure (Figure 3), which stores the total number of servers and their list. Attached to this structure are manipulation functions for initialization (`world_init()`), cleanup (`world_free()`), function for forced update of all servers (`world_check_updates()`) and a function to add a server (`world_add()`). The server add function performs either an update if the server is already known (stored in the structure) or an addition.

```
struct world_t
{
    world_server_t* servers;
    int count;
    int capacity;
};
```

Figure 3. Structure to store information about cluster

Information about individual servers is stored in a dynamic array without support for removing already known servers. This system is therefore not suitable for very dynamic environments (where servers are dynamically created and destroyed).

Information about single servers are stored in the structure shown in Figure 4, which serves as a wrapper around the internal data structure for storing server info `server_info`. This structure also stores the update server function (if such is provided) and additional information that is used for scheduling.

Current implementation does not include any specific update function implementations and relies completely on the information received during the scheduling cycle.

```

struct world_server_t
{
    server_info* info;                // server info
    int (*update_func)(world_server_t*); // update function
pointer
    prev_job_info *last_running;      // fair-share information
    int last_running_size
    unsigned int is_down :1;
};

```

Figure 4. Structure to store information about server

The data structure also does not contain the `last_update` value that will be probably needed for future scheduling implementations (information about servers that was downloaded long ago will definitely not be relevant). Current implementation only allows storing of the off-line flag. The `is_down` attribute is set to 1 (true) if a communication error occurs and set back to 0 (false) when the scheduler receives a new wakeup command from the server.

Attributes `last_running` and `last_running_size` are used for fair-share calculations.

4.2 Current FIFO Integration

After receiving the wakeup command from any server of the cluster, the scheduler reads the current server state and stores this information into the global structure using `world_add`. Afterwards, the `world_check_updates` function is called to update all servers that have a specific update function.

The main scheduling cycle is modified to support job moving. All jobs are still traversed using the `next_job()` function, which will after the initialization (`init_scheduling_cycle()`) return jobs in the order done by the current configuration (strict FIFO, fair-share, round-robin,...).

The biggest change is implemented for the case when function `is_ok_to_run()`, which is responsible for determining whether a specific job can run on a specific server, fails. If the job is stored in a global queue, it is not marked as `cannot run`, instead the scheduler will try to find place for the job on a different server. If such server is found, the job is moved.

This creates a problem with job priority (see Section 6.1.5).

4.2.1 Local vs. Global Queues

When working with M:N architecture, we are working with a concept of local and global queues. Local queues are used to store jobs that will never leave their home server. Global queues are a storage for jobs that can leave the server and therefore represent the base for cross-server scheduling. Global queues allow submits of jobs that require more resource then the current server can provide.

It might not be feasible to download information from all queues from every

server. It might be necessary to turn of information fetching about local queues on remote servers.

In the current setup, the scheduler is using information from local queues to calculate fair-share (which has to include local running jobs).

Support for server sided global queues has been accepted into Torque. Server was extended to support a new attribute for execute queues `is_transit`. Jobs submitted into queues with the `is_transit` attribute set to true are not checked against the resource limitations of the server.

The current implementation of `is_transit` only turns of the checking of max limits set on the server. Limits set on the queue and minimal limits (both for server and queue) are still checked (`chk_resc_limits()`). The same applies to other features like ACL.

The scheduler supports two additional options for global queues. One for determining the local server (`local_server`) and one to determine if the remote local queues (queues that are local but do not belong to the local server) should be ignored (`ignore_remote_local_queues`).

4.2.2 Job Moving

If the job is waiting in a global queue, it can be moved between queues or servers. Server responds to the message `PBS_BATCH_MoveJob`. If the move is local, server simply removes the job from one queue and inserts it into another (at the designated place).

If the move is cross-server, server starts the standard communication for job move (`req_quejob()`). After the job is successfully moved to another server, the destination server send back the information about the move (`issue_track()`) to the original server. The original server then stores this information into its internal structures (`server::sv_track`). This information can be later used to determine the current location of the job.

The receiving side does authenticate the job owner using `ruserok` (see Section 3.4.5).

The current implementation suffers from a problem with non-atomicity of the move-run process. If the server offers free capacity, all running schedulers will try to move jobs to this server, causing big move traffic (see Section 6.1.6).

4.3 Fair-share

The original fair-share was based on updating a list of running jobs (and recalculating the time difference).

List was updated on the end of each scheduling cycle (using the list of jobs the scheduler managed to run) and on the start of new scheduling cycle this list was compared to the current running job list of the server. Each match (job in both lists) meant that the job was running during the time scheduler was not active and therefore the fair-share information needs to be updated (see Section 3.1.1).

When working with multiple servers we need to store and update the list of jobs for each server separately. If we wouldn't the scheduler would just overwrite

this information on each scheduling cycle (server A is running different jobs then server B).

The fair-share calculation itself didn't need any modification. We still calculate fair-share globally, therefore the whole data structure can stay intact.

Problem that remains is desynchronization when scheduler outage appears. When running multiple schedulers, we want to remain in the state where all schedulers contain the same fair-share information. When a scheduler is offline, it does not update its fair-share information. Specifically it will miss the jobs that finished during the downtime (jobs still running will be caught in the next loop). This is a feature of the original Torque, see next subsection for implemented and proposed improvements.

4.3.1 Fair-share on completed jobs

Fair-share was slightly modified to calculate more precise values.

Original fair-share only checked the list of running jobs. The current implementation checks all jobs stored on the server. Therefore if a job ended in between the scheduling cycles it will be still recorded.

This modification will still miss jobs that started and terminated during the downtime.

4.3.2 Possible Future Extensions

The current extension does work with separate fair-share information for each scheduler. In a perfect setup with no downtime this solution will generate identical fair-share values on each scheduler.

If the frequency and length of downtimes will be small, we will only encounter small differences in fair-share values (schedulers will lose information about jobs that started and terminated during the downtime, or jobs that terminated during the downtime and the server already removed them for the job list).

But even small differences might cause problems for the scheduling logic, therefore the following ideas offer solutions with better potential to deal with downtimes.

- If decay will be set to low value, all fair-share information will quickly converge. Each decay represents halving of the old values, that also means all errors will be halved. If we will only encounter small differences and the decay timeout will be set to a short period, we can safely ignore the differences.
- Server is generating accounting information. This information contains everything that is needed to calculate fair-share. Therefore we can calculate fair-share completely externally and let schedulers only read the fair-share information in periodic intervals.
- We can also use the brute-force approach. If the server will store information about all terminated jobs (for certain time period) we could calculate complete fair-share information on each scheduling cycle. Each stored job (including those that already finished) contains information about used CPU time, the total run time and consumed memory. This information is provided by the master node the job was assigned to.

Global Fair-share

One remaining question is the problem of global fair-share application. We are calculating global fair-share (for all servers), but its usage is always local.

Model situation:

- Users: A, B, C, D
- Fair-share priority: $A > B > C > D$
- Server X: Jobs from users A and B
- Server Y: Jobs from users C and D

Scheduler will run job from user A on server X and job from user C on server Y. User B has to wait although he has higher fair-share priority than user C.

4.4 Nodespec Support

The original implementation of the FIFO scheduler did not include full nodespec support. Scheduler was simply trying to run all jobs on the server by sending run commands to the server without specifying the target node(s).

This causes the original scheduler to generate a huge amount of unnecessary server traffic, because it keeps requesting job runs that cannot be accommodated.

We will require the scheduler to decide what jobs will be run and where. This decision making process is needed for both the M:N architecture (decisions about job movement) and to eliminate server overloading.

Support for nodespec is concentrated in the `check_nodespec()`, which recursively filters and searches nodes that are suitable for the job. If such nodes are found, they are marked (`struct node_info::temp_assign`).

Our current implementation only supports one counted property, which is number of processes (`np`).

4.5 Server-Scheduler lock

When working with multiple schedulers per server we are rapidly increasing the amount of differences between the scheduler's idea about the cluster state and the actual cluster state. When trying to find a suitable node for a job, some other scheduler can start several different jobs on the server.

The biggest problem here is the fact that server is working with single commands that are atomic. Switching the atomicity to the scheduling cycle would solve most of the problems.

This approach does not solve problem with node sharing. If node sharing is enabled, nodes can change their state without the influence of the server.

4.5.1 Locking logic

Each message received through the `PBS_BATCH` protocol is checked against the lock. If the server is currently locked and the message came from a different source than the lock owner, the server will check the lock timeout. If the lock timeout is not up yet, the message is rejected (or passed through or delayed). To acquire and refresh the lock, the scheduler will use the `PBS_BATCH_SchedulerLock` message.

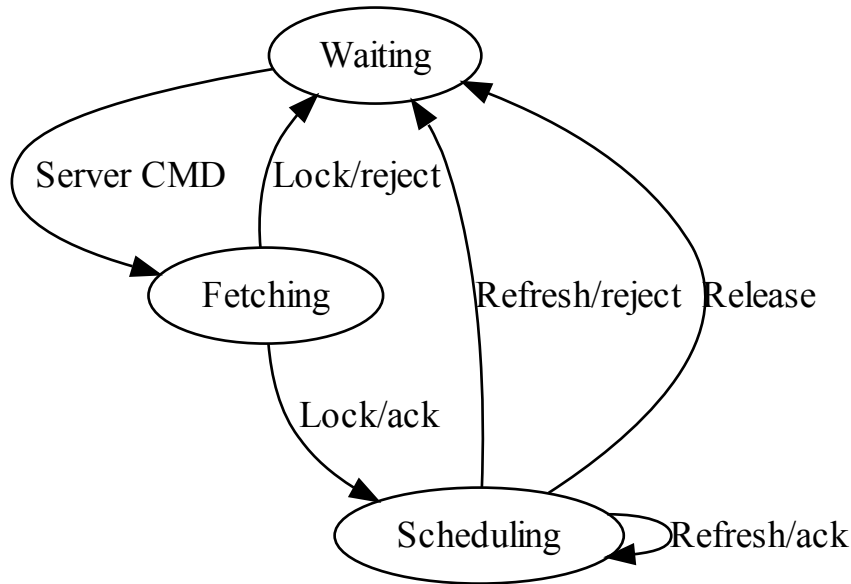


Figure 5. Proposed lock lifecycle

4.5.2 Splitting the PBS_BATCH protocol

We can't block all server commands during the duration of the scheduling cycle (which can be very long). All we want is mutual exclusion of schedulers. Therefore we will split the PBS_BATCH protocol into several groups with different approaches.

Indifferent commands

Indifferent commands are commands that do not affect the server state. Specifically, they do not change the state of jobs the scheduler might be currently scheduling. Such commands are not checked against the lock and are immediately processed. This includes most commands generated by user tools (qstat, qnodes, qsub...).

The following commands work with jobs in other than QUEUED state. Scheduler only works with jobs in QUEUED state and therefore we can safely let these commands pass.

PBS_BATCH_StageIn,	PBS_BATCH_JobObit,	PBS_BATCH_JobObit,
PBS_BATCH_SignalJob,	PBS_BATCH_Rerun,	PBS_BATCH_MessJob,
PBS_BATCH_CheckpointJob,	PBS_BATCH_Commit,	PBS_BATCH_RdytoCommit,
PBS_BATCH_jobscript,	PBS_BATCH_ReleaseJob,	PBS_BATCH_QueueJob

Read only commands do not change anything, they can pass as well.

PBS_BATCH_JobCred,	PBS_BATCH_LocateJob,	PBS_BATCH_SelectJobs,
PBS_BATCH_SelStat,	PBS_BATCH_StatusJob,	PBS_BATCH_StatusQue,
PBS_BATCH_StatusNode,	PBS_BATCH_StatusSvr,	PBS_BATCH_Rescq

These commands do work with jobs in QUEUED state, but in such way that it shouldn't collide with the scheduler.

PBS_BATCH_TrackJob, PBS_BATCH_AuthenUser, PBS_BATCH_ReleaseResc

Standard commands

Standard commands change the state of the server, jobs or nodes. Major characteristic of these commands is that they reserve resources.

These commands cannot be processed while there is a scheduling lock on the server.

PBS_BATCH_RunJob, PBS_BATCH_AsynrunJob, PBS_BATCH_ReserveResc

High priority commands

High priority commands are commands that change the server state, jobs or nodes, but their processing takes priority.

These commands will break any lock on the server.

PBS_BATCH_Shutdown

Delayed commands

These commands change the state of the server, but because they are used by user tools, we cannot just reject them. The following commands are added to a queue which is processed immediately after the lock is released or the lock is timed out.

The following commands remove jobs from the queued state, we might want to add them to high priority commands.

PBS_BATCH_MoveJob, PBS_BATCH_DeleteJob, PBS_BATCH_HoldJob

These commands encapsulate many different actions that can be both indifferent or can change jobs in queued state.

PBS_BATCH_Manager, PBS_BATCH_ModifyJob, PBS_BATCH_AsyModifyJob

The last command is order job. This command is only used by one user tool. It probably doesn't matter where we put it.

PBS_BATCH_OrderJob

4.5.3 Timeout settings

Setting the right timeout on the lock is a delicate problem.

We have two opposing problems. If we set the timeout to high, then we are running into the risk that a server will be blocked for a very long time when a crash/disconnect occurs. If we set the timeout to low, we will run into the problem of competing locks, where no scheduler can finish its scheduling cycle because his lock is always timed out and stolen by another scheduler.

We could lower the total time when one scheduler is blocking the server by setting the scheduling cycle to a fixed amount of iterations (only try the first 50 top-priority jobs).

Problem with competing locks could be solved by splitting the timeouts into two separate values. One for lock acquire (which should be longer, because after the lock is acquired, the scheduler starts with initialisation) and lock refresh timeout (which could be shorter).

4.6 Low Latency Scheduler

The original scheduler core does process received commands sequentially. In an infinite loop the scheduler sequentially accepts new connections and reads the command. This command is immediately handled over to the main scheduling cycle.

This approach generates several problems. Server will be blocked for the time the request will be processed. Server will only wait for the connection for a very short time (timeout), but even then, if we have several schedulers talking to the server and many scheduler commands generated, the server will be unnecessary slowed down by waiting. This can lead up to server starvation, where one server keeps timeouting other server.

4.6.1 Implemented Modifications

The base scheduler cycle was extended with two sub-loops.

First one is responsible for reading all commands waiting to be processed. Each connection is accepted and a command is read and stored into a special structure unique for the server. Commands are stored as flags, therefore each command is only recorded once. For each server, only one open connection is kept active. If a new command is received from a server that already has an active connection, the new connection is closed immediately after the command is received.

Second cycle is responsible for spawning the scheduling process for servers that sent a scheduling command. Supported is both the old system, when the scheduling cycle is run for each unique command and the new one, when the scheduling cycle is only run once for each server with a parameter being a set of commands received.

Structure for Command Storing

Individual commands received from servers are stored in a special structure (`struct commands`), which contains flags indicating the presence of each distinct command and also containing the IP address and the port to which these commands belong.

Structures are stored in a static array (`commands`), which is mapped 1:1 on the array of current connections (`connection`). Scheduling algorithm of the FIFO scheduler is using the Libifl library for communication purposes, all functions from this library are using the index in the array of current connection.

To maintain simplicity and future extensibility, we still store the index of the currently processed connection to the `connector` variable.

4.6.2 Future Modifications

These modifications represent a second stage of the low latency scheduler, which are still waiting for implementation.

Future modifications are designed to completely split the command receive cycle and the processing cycle. This extension should completely eliminate problem of server blocking (each connection should be accepted in the matter of milliseconds).

In case of full asymmetric setup, we can reach the limits of connection per second. This might be solved by limiting the number of connections on the server side.

The reading and processing cycle are clearly separated. The only shared variable is `connector`, which is storing the currently processed connection index (by the processing cycle) and array item pointed by the index. If we will only spawn one processing cycle in parallel, we don't have to worry about any data sharing problems. The reading cycle will simply skip the item pointed by the index stored in the `connector` variable and will check whether the parallel running scheduling cycle already finished. If it did, it will simply set the `connector` to a new value and start a new scheduling cycle (round-robin).

Tearing the scheduling cycle into a separate thread will probably not be trivially possible. Separate scheduling cycles still need to share state.

We could completely remove all state sharing by switching to scheduler initialized communication and reading information from all server at each scheduling cycle.

4.7 Queue Ignoring

For testing purposes and to support better scheduler configuration, a new support for queue ignoring (on the scheduler side) has been implemented into Torque. Queues to be ignored are specified in the configuration file in the following format: `ignore_queue: name`.

Implementation is using the already existent code that skips queues that are not set to run jobs. The queue and jobs in it are not completely ignored (information about each of the jobs is still pulled from the server), but jobs in such queues are marked as *cannot run*. This state ensures that the following parts of the scheduler logic will ignore this job (independent on the current scheduler settings).

5 Big Jobs

Because we are splitting the world into separate autonomous servers, we have to solve a new problem. How to run big jobs that need nodes from several servers? Such jobs don't even have to be extremely big. If a job requires several nodes with very specific properties, it can easily happen that such nodes cannot be allocated in one server.

We have several concurrent (and competitive) schedulers running. Therefore when starting a job using remote nodes (nodes that do not belong to the scheduled server) we have to ensure that these nodes won't be assigned to some local job.

We can't rely on a stable world state, because the schedulers are not synchronized and can even be asymmetric. View of the world that each scheduler keeps can be very different (see Section 6.2.1).

Competition for nodes can be solved using reservations, but reservations introduce different problems. Servers allow node reservation using nodespecs, but these reservations only work for whole nodes (no support for CPUs). Schedulers would have to keep information about current reservations and their mapping to jobs.

If we would like to use reservations we would soon hit problems with typical resource deadlock. We could introduce new job-substates QUEUED – QUEUED-ALLOCATING (server modification).

5.1 Possible Solutions for Big Jobs Problem

All following variants require equivalent logic on the scheduler side. This part of implementation is not documented here, because it simply has to match the selected variant.

5.1.1 Option 1: One Node Share by Multiple Servers

Nodes can be currently shared between servers (support added 03/2008). One node can belong to multiple servers and then sends all IS_PROTOCOL message to all such servers.

Each server could share some part of his nodes with other servers. The specific architecture is to be decided, because the current implementation offers relative freedom, including asymmetric architectures.

From the scheduler point of view, each server has more nodes and therefore running big jobs remains a local task, within one server that offers enough local and shared nodes.

Node names consist of local name and hostname, therefore it is no problem to determine which nodes are local to the server and which are shared. It would be useful to keep a side information about the level of sharing for each node (how many servers share the node). This value could be dynamically calculated on the scheduler side, but because such operation would require to check all nodes in all servers its not very feasible. This value could be used for preference assignments. Scheduler might prefer to run jobs on least shared nodes to minimize problems with sudden node state changing and also to prevent blocking big jobs which are

dependent on the availability of these nodes.

This variant is mostly about scheduler algorithms. Support for node sharing is already present. This code however lacks the stability required for production environment.

5.1.2 Option 2: All Nodes in One Big Server

We could go a step back and bring back the old architecture, but only for big jobs. We could create one additional server for big jobs that will have no local nodes, only shared nodes from other servers. This setup should not pose a "single point of failure" problem because big jobs cannot work without the connectivity between nodes anyway. Such server would serve the sole purpose of running big jobs that cannot run on any of the separate servers locally.

Biggest plus of this setup is that all functionality is already present and the overall simplicity of proposed architecture.

This solution would also provide a possibility to run a completely custom scheduler designed solely for the purpose of scheduling big jobs. This scheduler can be much slower than any other scheduler in the system and can work with specific information like geographic location. There is no chance of scheduler collision because the scheduler is the only one talking to the server.

Yet the scheduler would still have to cope with the fact that single nodes can change state during scheduling. This is something that even the simple FIFO scheduler has to work with, but the global scheduler would have to work around the fact the local scheduler have much faster access to their servers and nodes. It should be noted that a separate scheduler will bring back a lot of problems that are mostly associated with hierarchic scheduling, but is not a requirement for this variant.

Same as option 1, this option is mostly about scheduler algorithms.

5.1.3 Option 3: Job Running Using Multiple Servers

Last option is to run jobs using multiple servers.

Dynamic Node Sharing

To avoid most problems of node state changes, we can leave nodes local and only share them if a scheduler requests. Implementation would consist of two new IS_PROTOCOL messages IS_ATTACH and IS_DETACH, that would contain the host-name of the server, to which should the nodes connect. Server would generate this messages as a reaction to PBS_BATCH_SHARE and PBS_BATCH_UNSHARE messages (received from the scheduler), or we could implement this feature by communicating directly with nodes.

This approach will probably suffer from long run job latencies. Before each job run, the scheduler has to contact all servers that will participate in the job run that they need to share nodes (that requires currently unimplemented scheduler initiated connections). Only after all required nodes are shared, the scheduler can finally start the job run.

This communication can be implemented on server side (with scheduler sending command only to the main server), but such implementation will still suffer from the same latencies.

If a node is shared the local server could mark the node as full. This would be inefficient, but would prevent problems with node state changes.

Current implementation of node sharing doesn't allow dynamic adding and removing of servers (only clients). This features shouldn't be very hard to implement (only in the case of big server counts, we might need to switch to more effective data structures). Implementing new messages shouldn't be a problem either.

Direct Run Across More Servers

Command to run job has to be sent to the server, which has to job stored.

The list of nodes, required to run the job can reach the server from several channels. It can be sent by the scheduler, or taken from the job attribute (neednodes), or generated by the server itself. In all these cases the server is limited to his own nodes.

Direct running of parallel jobs would require modification of the server logic. The server would have to ignore the fact that it doesn't know what is actually happening on the remote nodes and simply pass the master node the list of all nodes.

Other servers (whose nodes will be participating on the job) would have somehow notify these nodes that they have a new sister (or each node would require a full list of all other nodes).

See working with remote jobs on nodes in Section 6.1.4.

6 Future Problems and Ideas

6.1 Problems to Solve

The following list contains encountered problems that were deferred.

6.1.1 Removing Jobs that Require More Resources then the Server Can Provide

Scheduler is responsible for deleting jobs that are too big to run on the server.

This feature was switched off, because a scheduler cannot determine if the current cached state of the cluster does actually correspond to the real state of the grid. There could be a temporary error or connection outage. Scheduler could also be configured in an asymmetric matter (where it only talks to a portion of the clusters).

It will be necessary to add some external monitoring to determine jobs that need to be erased.

6.1.2 Connection Initialized by the Scheduler

When sending commands to the scheduler, the initialization of the connection is done by the server. The server creates a new connection to the scheduler and then

sends a wakeup command through this connection. When the connection is established, the server itself marks the necessary flags on the connection, so that all following communication with the server can be done through this connection.

Support for connections initialized by the scheduler might simplify other features and its implementations should be relatively easy.

6.1.3 Scheduler-MOM Communication

RPP communication with nodes might lead to scheduler/node overloading, when we switch to M:N architecture.

This communication might cause cascade effects (one overloaded point in the clusters slows down the whole cluster). Due to the asynchronous communication model, these cascade effect should remain relatively localized, but it might be necessary to completely turn this communication off.

6.1.4 Foreign Jobs on Nodes

When sharing nodes, either on the server level, or just by giving the node a list of foreign sister nodes, we end up with a running job, that the nodes parent servers doesn't have in its database. This causes problems when calculating free resources on nodes.

6.1.5 Scheduling Cycle

The main scheduling cycle doesn't follow priorities strictly. Small jobs are preferred over big jobs, because when a big job fails to run, the main cycle continues until all jobs have been tried.

This allows good usage of resources, but can cause job starvation.

6.1.6 Problems with Job Moving

In the current implementation, job moving is done using the PBS_BATCH_MoveJob message. This message will cause a correct job move to another server, but because it is just a move, there is no resource allocation done on the remote server.

Scheduler does simulate the resource allocation in its internal structures, but using multiple schedulers can lead to a very active job movement. If resources are freed on one server, N schedulers have the potential to move N-times more jobs than the server can handle.

Reservation

We could allocate the required resources using reservations. Problem with reservations is typical resource deadlock. And once again, this feature would require the scheduler to be able to initialize new connections (see Section 6.1.2).

MoveAndRun

We could introduce new message MoveAndRun. This would be sent instead of move message and would not only ensure move but also a direct run of the job.

6.2 Future Ideas

This section describes possible future extensions and ideas to be implemented into the Torque system.

6.2.1 Sending the State

Scheduler could send its current idea about the server state together with every request (this node is free, that one is running 3 jobs on 5 CPUs, etc...). This would be specifically useful for commands like job run and job move.

Server would then check if the state does actually correspond to the reality. If not, the request would be declined.

It is unclear how would such setup work for M:N configuration where M:N are big numbers (5-10). It is possible that the request rejection rate would grow over usable limit.

6.2.2 Code Strengthening

We haven't discussed the code quality in this document. Torque suffers from its very long code history (some code parts are more than 20 years old).

While working with the code, it would be wise to dedicate a portion of the time to gradual code refactorization (at least on local scope). Code should be strengthened using unit tests and assertions.

Three different assertions were already added into the code:

`dbg_precondition(expr, comment)`, `dbg_postcondition(expr, comment)`, `dbg_consistency(expr, comment)` First two are designed to catch input and output requirements (NULL params), the third is designed to catch inconsistencies.

Another issue with the current code is IPv6 support. Torque contains an IPv6 branch with ongoing development effort, but there is no set timeline for the inclusion of this branch into stable Torque. We have developed patch-set adding IPv6 support to PBSPro, this patch-set could be ported with reasonable effort to Torque, too.

7 Feature Mapping

In this section, main features of PBSPro batch system, used in MetaCentrum environment, are enlisted, together with mapping to Torque features. Main goal is identify missing features, possible solutions for implementation and estimate time required for reimplementing all features required for first experimental installation and following full-featured installation.

Format of feature list is following:

— *PBSPro feature, with short explanation*

Torque feature which can be used, info if the same feature is available or missing etc.

7.1 Queues

In both systems, submitted, running and completed jobs are enlisted in queues. Each server can maintain a number of queues, each queue with different settings.

- *Different queue priorities, mainly used for prioritization in scheduling of jobs.*
The original FIFO scheduler supports different simple scheduling schemes. One of these schemes is queue priority scheduling. When implementing our own scheduling algorithm, queue priorities can be easily incorporated.
- *Limits on number of jobs enqueued, running for each queue*
Toque supports both limits on server and queues.
- *Limits on maximal and minimal required walltime, cputime ,and number of CPUs*
Torque does support the np parameter which has a bit different semantics then number of CPUs (when handling shared and exclusive cluster nodes). Maximal and minimal cputimes and walltimes can be set both per server and per queue.
- *Routing and execution queues, each queue enabled for scheduling separately.*
Full support in Torque.
- *Optional required property on nodes, only nodes with such property can be used by jobs in the queue.*
In PBSPro implemented as port of OpenPBS patch, needs to be ported to Torque.

7.2 Nodes

Computational nodes are managed by server which is monitoring node availability, status of node, status of jobs submitted to these nodes. Development version of Torque supports also nodes shared between several servers, such feature may be useful for multi-site jobs.

- *List of properties defined per node (architecture, OS flavor, location, dedication, special hardware/software available on this node)*
Torque has full support for properties without values.
- *Nodes optionally dedicated to queue*
Not supported in Torque.
- *Node available for subset of queues (using required_property feature)*
Not supported in Torque. required_property patch could be ported to Torque.
- *Nodes with specialized network with additional property defining network switch.*
Not supported in Torque.
- *Cluster nodes (in PBS-Pro terminology), both job-shared and exclusive*
Torque supports cluster nodes with the same semantics with the slight problem of CPU counting. Torque does not support CPU as a resource, but instead works with processes. This causes one slight problem with exclusive nodes (node is kept in free state until all processes are busy).
- *Nodes with property no_multinode_jobs (jobs requiring more than one node cannot be running in this node)*
Not supported in Torque.

7.3 Jobs

Jobs are submitted to queues, with description of requirements on resources (CPU, memory, scratch, licenses, ...).

- *User-defined/server_default/queue_default requirements on resources like CPU, memory, walltime*
Torque supports walltime and cputime limitations. Memory is not supported as a resource.
- *User-defined requirements on number and properties of nodes*
User can specify any amount of nodes and properties, but Torque does not support any counted resource (with the exception of NP).
- *Optional resubmit of job in case of compute node failure.*
Torque has rerunnable support.
- *Some implementation of job collections (arrays in PBS-Pro)*
Torque supports job arrays.
- *Some form of dependencies between jobs*
Torque has extensive support for job dependencies.
- *Optional email on job start/stop/fail*
Torque supports sending emails upon start, termination and abort.
- *Optional file staging (stage-in/out), stdout/stderr output transfer after job is finished.*
Full support in Torque.
- *Support both for batch and interactive jobs.*
Full support in Torque.
- *Prologue/epilogue scripts are started (with admin privileges) by PBS before/after job is running*
Full support in Torque.

7.4 Security

- *Kerberos authentication for job submit*
MetaCentrum patch, could be ported to Torque with reasonable effort
- *Kerberos/AFS credential renewal during job lifetime*
MetaCentrum patch, could be ported to Torque with reasonable effort
- *ACL for server or queue, based on IP(hostname) or username+IP(hostname)*
Full support in Torque.
- *Queue ACL using unix group*
Not supported by Torque.

7.5 Scheduling

Several large modifications to PBSPro scheduler were developed in MetaCentrum (virtualization support, better handling of starving jobs, better support for scheduling using more resources etc.). We expect that most of this work will be ported to Torque scheduler, which has roots in the same original code, but currently is much weaker in number of features.

- *“FIFO” – jobs sorted using queue priority, fairshare in one queue, job arrival time or job required time otherwise.*
Queues can be sorted by their priority, jobs can be sorted both globally and in their queues by fairshare and walltime.
- *Support for preemption (using our virtualization patch)*
MetaCentrum patch, could be ported to Torque with reasonable effort.
- *Support for large parallel jobs (using our virtualization patch)*
MetaCentrum patch, could be ported to Torque with reasonable effort.
- *Support for starving jobs*
MetaCentrum patch, could be ported to Torque with reasonable effort.
- *Node packing strategy, nodes with smaller number of free CPUs packed first*
Not supported in Torque.
- *One-to-one allocation for processor and job process, no over-subscription of nodes*
Torque works with processes semantics which are mapped 1:1 only for exclusive jobs. Extended scheduler implementation works with correct 1:1 mapping.
- *Understanding of network topology of clusters, network switches (all nodes of computation on one switch)*
Not supported in Torque.
- *Scheduler as separate process, with full control on decision on job startup and placement, minimal decisions on server side (easier for development of new scheduler)*
Scheduler has full control over the Server which will even allow many wrong job runs (like running another job on a node with exclusive job).

7.6 Resource Management

Resources required by jobs during submission and used by running jobs are monitored on several levels (node level on each node, global level for parallel jobs on server, external resources representing for example available licenses on external level etc.).

- *Maximal number of jobs for user or group, on server, queue and node level*
Only generic support (maximum limits for any groups), not enforced by the server.
- *The same for number of used CPUs*
Not supported in Torque.
- *Jobs with limits on resources (memory, CPU) enforced on nodes using ulimit*
Not supported in Torque.
- *Central monitoring for job-wide limits*
Supported, counted on master node.
- *External resources on node level, static and dynamic (currently used via pbs_cache)*
Can be ported to Torque.
- *External resources on server level (software licenses)*
Not supported in Torque.
- *Possibility to utilize advanced Linux features – cpusets, cgroups*
Torque supports cpusets.

7.7 User and Administrator Commands

Compatibility on user and administrator API and command-line level is expected, both systems were developed on top of original OpenPBS source-code and there was not large development on this part of code.

- *User command: qsub, qstat, qdel, qalter, qsig, qhold, pbsnodes*
Torque provides these commands with very similar interface.
- *C-library, usable for MetaCentrum web-interface*
Torque consists of base Libtorque library and binaries built upon it.
- *pbsdsh, pbs_attach for MPI jobs*
Torque supports only pbsdsh.
- *qmgr for administrators*
All server settings are managed using qmgr.

7.8 PBSPro Features not Used in MetaCentrum

Several features provided and advertised by PBSPro were not used in MetaCentrum. Therefore there is no need to reimplement them in Torque.

- *Advance reservations (not used due to bad impact on scheduling)*
Torque supports node reservations using nodespecs. This reservation system works only with full nodes.
- *Preemption (original implementation not usable without checkpointing)*
- *Node_group (packing of parallel jobs with network topology awareness) in current state of implementation*
- *Backfilling (not usable with not-precise jobs duration in job specification)*
- *Load-balancing on time-shared nodes*
- *PBS Pro as front-end for Globus (not working with latest versions of Globus)*
- *Cycle harvesting (we are not using desktop machines)*
- *PBS redundancy (backup server, but with shared spool directory on NFS, not usable across MetaCentrum)*
- *Prime/non-prime times, different scheduling in working-hours*

7.9 Modified/Developed Features:

Several features were developed in MetaCentrum as enhancements of PBSPro. Most of these features must be ported to Torque.

- *Kerberos support (Kerberos ticket check and encoding in qsub command, credential renewal on node level)*
- *Support for Magrathea and virtualization, must be ported to Torque*
- *Number of scheduler improvements, must be ported to Torque scheduler*
 - support for node reservation used by starving jobs, modification only is scheduler
 - pbs_cache support
 - mem/vmem resources
 - site_user_has_account
- *Number of stability improvements, must be ported to Torque server+mom*
- *Required_property for queues, inspired by OpenPBS feature, can be ported to Torque*

- *pbs_mom restart when new config/binary is available, inspired by Torque feature*
- *Completed jobs visible in qstat, inspired by Torque feature*

7.10 Summary of Main Problems

- As expected, Torque Scheduler does not fulfill all required features. Current implementation must be changed to get full control on decision on job startup and placement, several missing features must be implemented, features implemented by MetaCentrum in PBSPro must be ported for this scheduler (virtualization support, better support for starving jobs, pbs-cache support).
- Most deficiencies of Torque are based in the most typical Torque setup. Torque is usually used in a 1:1 installation with some big external scheduler. Resources and other limits are then managed on the scheduler side. We can do the same in our scheduler, but we also have to incorporate this logic into server.
- The most important problem currently is support for exclusive/shared cluster nodes vs. number of CPUs. Torque only supports processes semantics (not CPU); when a node is selected to have more than one process, exclusive/shared/free flags are not calculated correctly on the server.

8 Conclusion

The current job scheduling architecture for the MetaCentrum is already reaching its limits. We have therefore designed a new decentralized solution, which solves major problems we are experiencing. While decentralized hierarchic architecture is commonly used in many grids, we introduced an inter-cluster extension to the scheduling scheme. This will provide us with high quality scheduling through the MetaCentrum, while maintaining good 3rd party support through gateways.

Our proposal is based on the Torque batch system that offers a good base for the requested features while providing a familiar interface for both the users and administrators.

We have mapped the current state of development, including an overview of areas where development is still needed. Most still unimplemented features are concentrated in the following areas:

- porting features from PBS-Pro
 - all features implemented as an extension of PBS-Pro must be ported into Torque
 - this work will rely on the common code base of PBS-Pro and Torque
- re-instating server into the coordination role
 - our architecture requires the server to verify all run requests
 - each resource requirement of the request has to be checked against current available resources
- implementing support for big jobs running over several physical clusters
 - torque already includes preliminary support for node sharing
 - available support has to be extended and integrated with the rest of the system

- implement scheduler logic
 - scheduler logic has to match the capabilities of the server
 - all modifications to the server and mom components have to be matched in the scheduler logic
- provide transparent transition
 - transition to Torque has to be provided as a transparent update with no changes in end user functionality
 - final behaviour of Torque will mimic the current functionality of PBS Pro

This work outlined the plans for the year 2010. We are expecting to have a production ready environment using Torque and inter-process scheduling algorithms by the end of 2010.

References

- [1] FOSTER, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, p. 2–13, 2006.
- [2] JACKSON, D.; SNELL, Q.; CLEMENT, M. Core Algorithms of the Maui Scheduler. In *Proceedings of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.
- [3] LITZKOW, M.; LIVNY, M.; MUTKA, M. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing. Systems*, 1988, p. 104–111.
- [4] ZHOU, S. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, 1992.
- [5] BARHAM, P.; DRAGOVIC, B.; FRASER, K.; HAND, S.; HARRIS, T.; HO, A.; NEUGEBAER, R.; PRATT, I.; WARFIELD, A. Xen and the Art of Virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*. October 2003.
- [6] SOLTESZ, S.; POTZL, H.; FIUCZYNSKI, M. E.; BAVIER, A.; PETERSON, L. *Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors*. April 2007. Available online⁸.
- [7] RUDA, M.; DENEMARK, J.; MATYSKA, L. Scheduling Virtual Grids: the Magrathea System. In *Proceedings on the 3rd International Workshop on Virtualization Technology in Distributed Computing*, Reno, USA. Article no. 7. ACM, 2007.
- [8] YOO, A.; JETTE, M.; GRONDONA, M. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*. Lecture Notes in Computer Science, vol. 2862, p. 44–60, Springer-Verlag, 2003.

⁸ <http://wireless.cs.uh.edu/presentation/2006/07/Container.pdf>

- [9] RAJIC, H.; BROBST, R.; CHAN, W.; GARDINER, J.; HAAS, A.; NITZBERG, B.; TOLLEFSRUD, J. *Distributed Resource Management Application API Specification 1.0*. Document GFD.22, The Open Grid Forum, DR-MAA Working Group, 2003.
- [10] ANDREETTO, P. et al., CREAM: A simple, Grid-accessible, Job Management System for local Computational Resources. In *Proceedings of XV International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, February 13-17, 2006, Mumbai, India. Macmillan, p. 831-835.
- [11] KORKHOVA, V. V.; MOSCICKIB, J. T.; KRZHIZHANOVSKAYA, V. V. Dynamic workload balancing of parallel applications with user-level scheduling on the Grid. *Future Generation Computer Systems*, vol. 25, no. 1, 2009, p. 28-34.
- [12] HUEDO, E.; MONTERO, R. S.; LLORENTE, I. M. The GridWay Framework for Adaptive Scheduling and Execution on Grids. *Scalable Computing – Practice and Experience*, vol. 6, no. 3, p. 1-8, 2005.
- [13] CZAJKOWSKI, K.; FOSTER, I.; KARONIS, N.; KESSELMAN, C.; SMITH, M. S.; TUECKE, S. A resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*. 1988, p. 62–82.
- [14] HENDERSON, R.; TWETEN, D. *Portable Batch System: External reference Specification*. NASA, Ames Research Center, 1996.
- [15] LAURE, E.; HEMMER, F.; PRELZ, F.; BECO, S.; FISHER, S.; LIVNY, M.; GUY, L.; BARROSO, M.; BUNCIC, P.; KUNSZT, P.; DI MEGLIO, A.; AIMAR, A.; EDLUND, A.; GROEP, D.; PACINI, F.; SGARAVATTO, M.; MULMO, O. Middleware for the next generation Grid infrastructure. In *Computing in High Energy Physics and Nuclear Physics (CHEP)*, 2004.