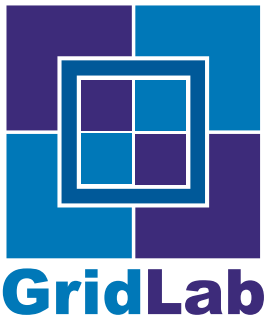


IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

# Result Evaluation

| | |
|---|---|
| Author(s): | Jason Maassen, Rob V. van Nieuwpoort, Andre Merzky, Thilo Kielmann |
| Document Filename: | GridLab-7-EVAL-0005-ResultEvaluation |
| Work package: | WP7: Adaptive Components |
| Partner(s): | Vrije Universiteit (VU) |
| Lead Partner: | Vrije Universiteit (VU) |
| Config ID: | GridLab-7-EVAL-0005-1.0 |
| Document classification: | IST |

**A**bstract: This document provides an evaluation of the adaptive components developed within the GridLab project, with respect to feasibility and achieved performance gain, based on the use cases as described in the report GridLab-7-UCR-0001-2.0.

# Contents

# 1 Introduction

In deliverable GridLab-7-CORE-0003-1.0, we described the design and implementation of the Delphoi system; the basic infrastructure that is needed to implement adaptive components. The Delphoi is currently deployed on the GridLab testbed, and gathers the machine and network information necessary to address the use cases, as described in deliverable GridLab-7-UCR-0001-2.0. It also provides a user interface to conveniently access the information gathered.

In the following sections, we will give a (brief) description of the design and implementation of the components developed for each of the use cases, and then evaluate the performance gains achieved by using these components.

# 2 Adaptive Use Cases

Adaptive components for addressing the use cases as described in the document GridLab-7-UCR-0001-2.0 have been implemented as deliverable GridLab-7-COMP-0004-1.0. In this section, we briefly describe the components and evaluate the performance gains that can be achieved.

## 2.1 Resource Selection for Parallel Triana Units

### Scenario

Triana programs are formed by so-called *units*, connected to a flow graph, through which data items flow from sources to sinks. Each unit performs some computation to process the data. Units may have multiple inputs and outputs, and they may be composed from other (simpler) units. Units can be parallelized in a transparent way. An input stream of a parallel unit can be split into blocks, which can be handled independently in parallel by multiple instantiations of a sequential unit. When the input blocks have been processed, the resulting outputs are combined again into a single output stream.

In a Grid environment, a Triana flow graph is supposed to run on a distributed set of machines. Parallel units need to be mapped or replicated onto a set of those machines for execution. The problem is to optimize the number of machines, so the overall system computes fast enough to meet the demands of the application.

This adaptation problem can be solved by continuously monitoring the performance of the machines that perform the computation. By gathering this information in a central location, the total performance of the distributed application can be estimated. This total performance is then compared to a user-specified target to determine if the current set of allocated machines is adequate, if more machines are needed, or if less machines would suffice. This results in the following feedback loop:

1. measure performance of each unit
2. determine total performance
3. compare total performance to target
4. adjust number of machines used (if required)
5. repeat the entire process

This use case is implemented in the *Quality of Service Adaptive Component*, which can be downloaded from the GridLab web page, and has been described in more detail in the deliverable GridLab-7-COMP-0004-1.0. This adaptive component consists of a library that contains the necessary logic to perform the feedback loop described above.

### Results

To evaluate the quality of service adaptive component we have developed two *task farming* applications. This type of application is widely used in grid environments, and its behavior is very similar to the Triana scenario described in the use case.

In general, a task farming application consists of a *server*, which produces *tasks* (or *jobs*), and one or more *workers* which retrieve tasks from the server and process them. (Triana applications often have a similar structure, where one unit generates data, which is then processed by other, 'worker' units.) We will now describe the applications in more detail.

**Simulation** To be able to easily evaluate the behavior of the adaptive component in different environments, our first application is a task farming *simulation*. This simulation consists of an *application manager* and a set of *simulated workers*. When the application manager is started, it creates an adaptive component and informs it of the application's target performance. It also creates a single simulated worker and forwards the worker's unique identifier to the adaptive component. The adaptive component then uses the Mercury monitoring system developed by WP-11 to retrieve performance information about this worker.

Although the workers do not perform any real computation, they can be configured to simulate the circumstances on a real grid. The performance they report to the adaptive component can be made to vary, both between workers and over time. During the run of the application, the adaptive component will continuously use the monitoring system to gather the performance data provided by the workers, and notify the application manager when the number of workers needs to be adjusted in order to reach the specified performance target. The application manager then adds or removes workers, and informs the adaptive component of the changes. From the viewpoint of the adaptive component, this simulation behaves in exactly the same way as a 'real' task farming application. It does not notice any difference. We will now discuss the results obtained for various worker simulations.

Figure 1 shows the result of the first simulation. In this simulation, all workers have the same performance, which does not vary over time. Therefore, this simulation corresponds to an application running on a homogeneous system, where machines are reserved exclusively for a single job.

The bottom graph in the figure shows a single line, the total number of workers allocated. The top graph shows three lines: the *target throughput* which the adaptive component tries to obtain, the *total worker throughput* which is generated by the workers, and the *estimated worker throughput* which is the throughput estimated by the adaptive component based on the information provided by the workers. Note that in real applications, the total worker throughput is usually not known, and only the estimated worker throughput is available. Because the workers only report their performance to the adaptive component after they have completed a task, the estimated throughput lags behind the total throughput.

In this simulation, each worker requires 10 minutes to process a single task. Therefore, 10 workers are needed to obtain the target throughput of 1 task per minute. As Figure 1 shows,

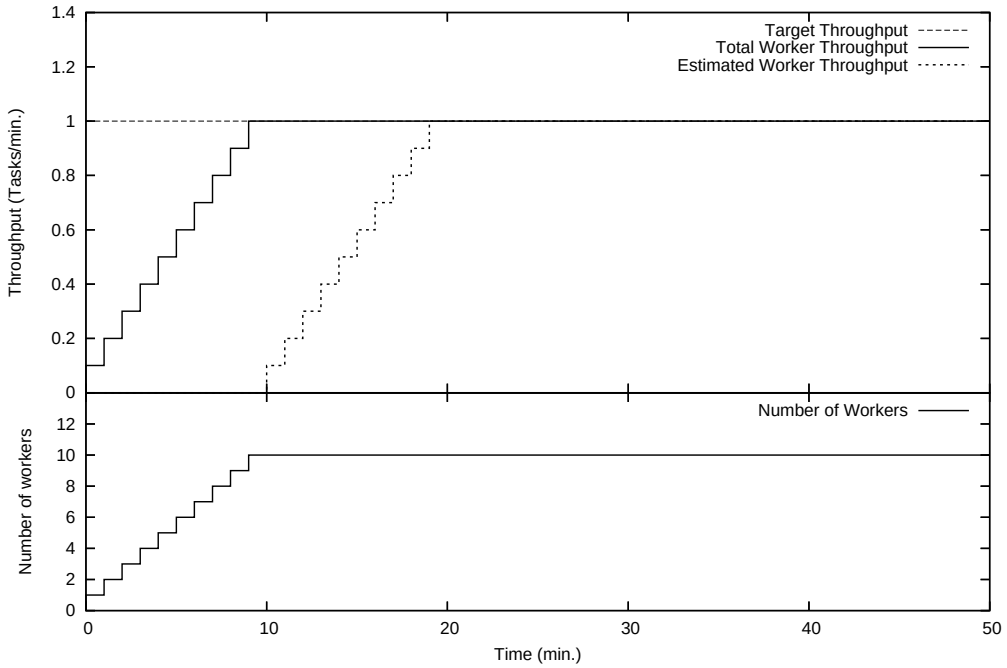


Figure 1: Performance adaption with homogeneous workers with a fixed performance.

the adaptive component is not able to provide a performance estimate for the first 10 minutes of the run, because no worker information is available yet. However, since the adaptive component knows that a task should be processed every minute, it can still signal the application manager to add a new worker every minute. This way, the application will at least keep up with the tasks until more information becomes available.

After 10 minutes, the first worker has finished a task and provides performance information to the adaptive component, allowing it to produce a first performance estimate. Since this worker will start on its second task, the adaptive component concludes that it is not necessary to add a new worker. From that moment on, performance information will come in once a minute, and no workers will be added. After 20 minutes, the adaptive component has performance information about all workers. Since the estimated worker performance is then equal to the target performance, no further resources need to be added or removed for the rest of the run.

As the results in Figure 1 show, the adaptive component is able to quickly increase the number of workers to the required level, even if there is no performance information available yet.

Figure 2 shows the result of the second simulation. In this simulation, the performance varies amongst the workers, but it does not vary over time. This corresponds to an application running on a heterogeneous system, where machines are reserved exclusively for a single job.

Each worker requires between 5 and 10 minutes to process a single task. The exact performance is randomly chosen when the worker is created, but remains constant for the rest of the run. Therefore, at most 10 workers are needed to obtain the target throughput of 1 task per minute. Like in the previous simulation, the adaptive component is not able to provide a performance estimate for the first part of the run, and simply adds a new worker once a minute. After 9 minutes, the first performance information is received and no more workers are added.

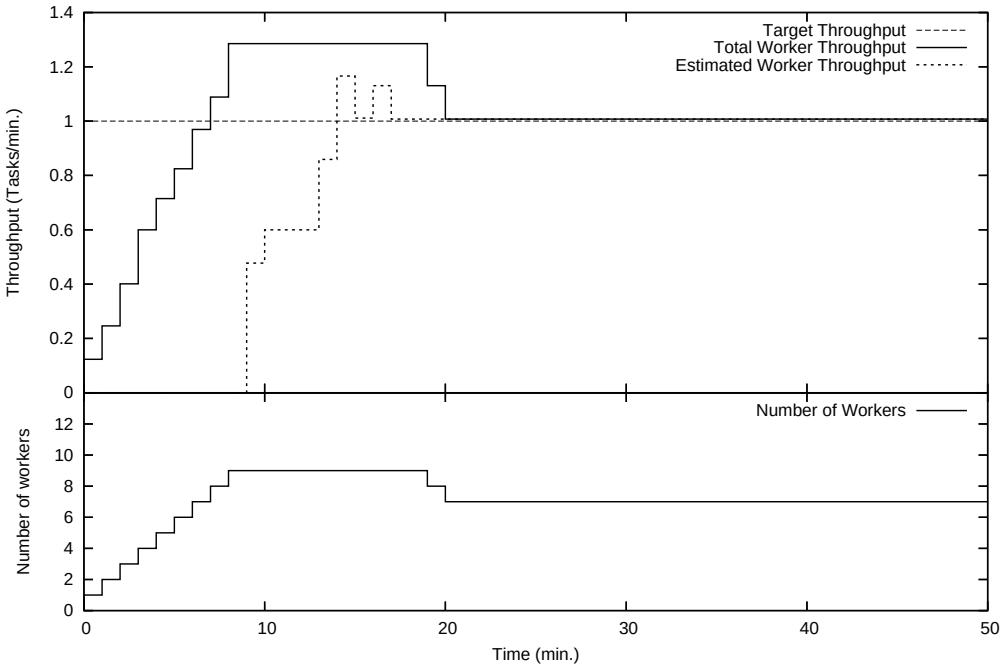


Figure 2: Performance adaption with heterogeneous workers with a fixed performance.

After approximately 15 minutes, the adaptive component has gathered enough information to determine that too many workers are active. It then selects a worker to be removed and instructs the application manager to do so. After the application manager acknowledges that it will indeed be removed, the adaptive component removes the worker's share from the performance estimate. Note that the worker is not removed immediately, but first finishes its current task. Therefore, it does not leave the computation until the 19th minute.

The results in Figure 2 show that the adaptive component is able to handle a variation in worker performance, and that it will reduce the number of workers if necessary. An extreme example of this will be shown in the next experiment.

Figure 3 shows the results of an experiment where 9 'slow' workers are allocated, followed by one 'fast' worker. This last worker is fast enough to achieve the target performance on its own. Since the adaptive component has a preference for a small number of fast workers. It will remove each of slower workers, until only the fast one is left.

The results of the final simulation are shown in Figure 4. There, the performance varies amongst the workers, and also varies over time. This simulation corresponds to an application running on a heterogeneous system, where machines are running multiple jobs simultaneously.

As with the previous simulations, the adaptive component starts by quickly increasing the number of workers. After some 20 minutes, it decides there are too many workers and releases two of them. For the rest of the 500 minute run, the number of workers is fairly stable, although their number changes occasionally. This experiment shows that the adaptive component works in an environment with dynamic worker performance, such as a grid.

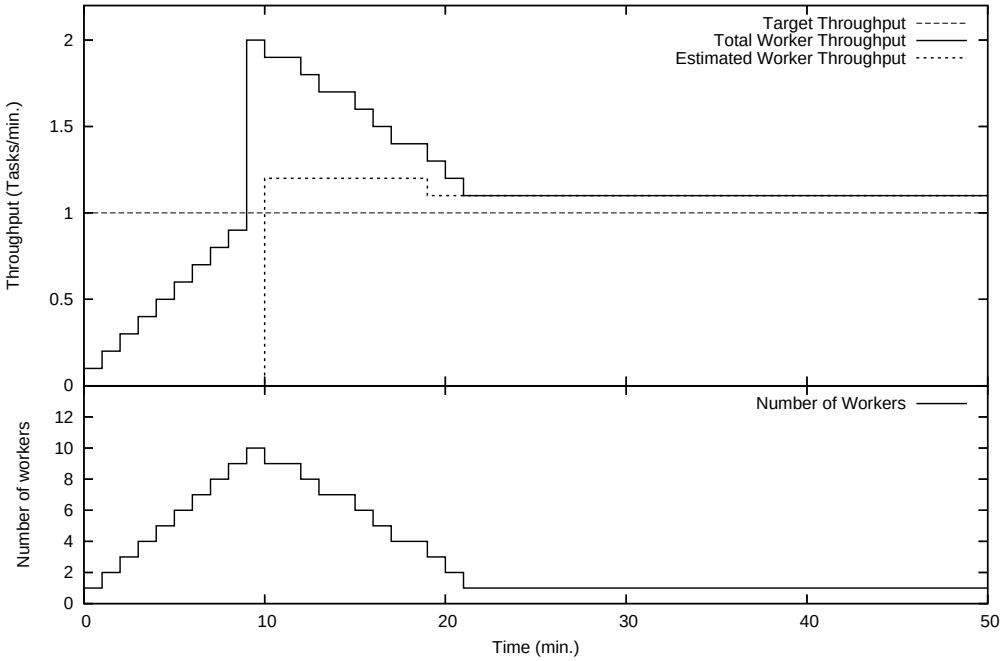


Figure 3: Performance adaption with heterogeneous workers with a fixed performance (extreme example).

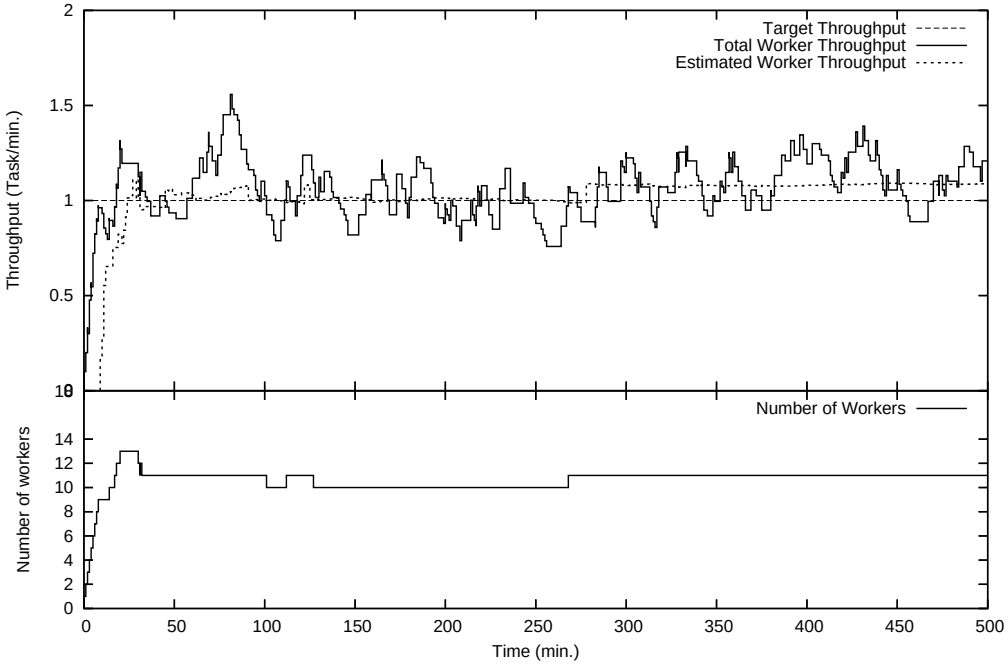


Figure 4: Performance adaption with heterogeneous workers with a varying performance.

**Search application** To further evaluate the quality of service adaptive component we have developed a real task farming application. We have chosen to use a simple search application, a variation of the *Traveling Salesman Problem*. This application tries to find the shortest route to visit a number of cities by exhaustively searching all possible routes.

When the application starts, the server generates a large number of tasks, each containing a partial route. The workers then retrieve these tasks from the server and compute all possible completions of the route. Like in the simulation described above, this application also uses an *application manager*. This manager sets a performance target for the application, and uses the adaptive component to monitor its performance. To create new workers, the application manager submit jobs to the GridLab testbed using the Java implementation of the GAT (developed by WP-1).

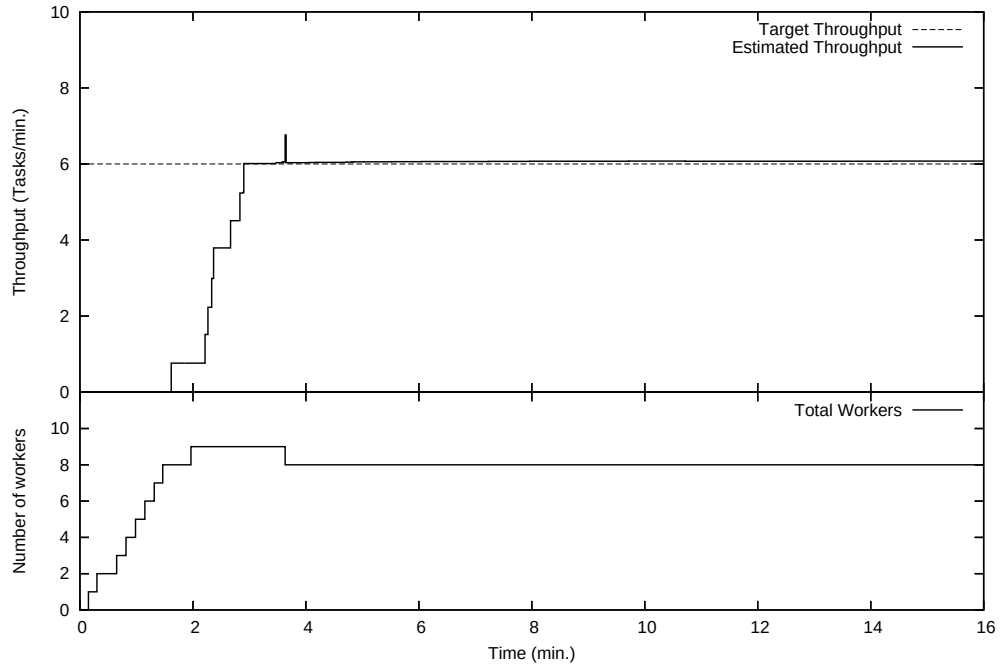


Figure 5: Performance adaption on a task farming application on the fs0 cluster.

Figure 5 shows the performance result of the application in homogeneous setting. All workers are submitted to run on nodes of the fs0 cluster (Amsterdam, The Netherlands). As explained before, the *Total Worker Throughput* is not available in real applications. Therefore, only the *Target Throughput* and *Estimated Worker Throughput* are shown.

As shown in Figure 5, the target throughput is set to 6 tasks per minute. Each worker requires almost 80 seconds to process a task, so 8 workers are necessary to achieve the target throughput. Initially, a extra 9th worker is allocated. This is caused by the short scheduling delays when new workers are started. Due to these delays it takes longer for the first performance information to be produced by a worker. The adaptive component compensates for this by creating an extra worker.

Figure 6 shows the performance result of the application running on the GridLab testbed. Part of the application is still running on the fs0 cluster, but new workers are also submitted to a

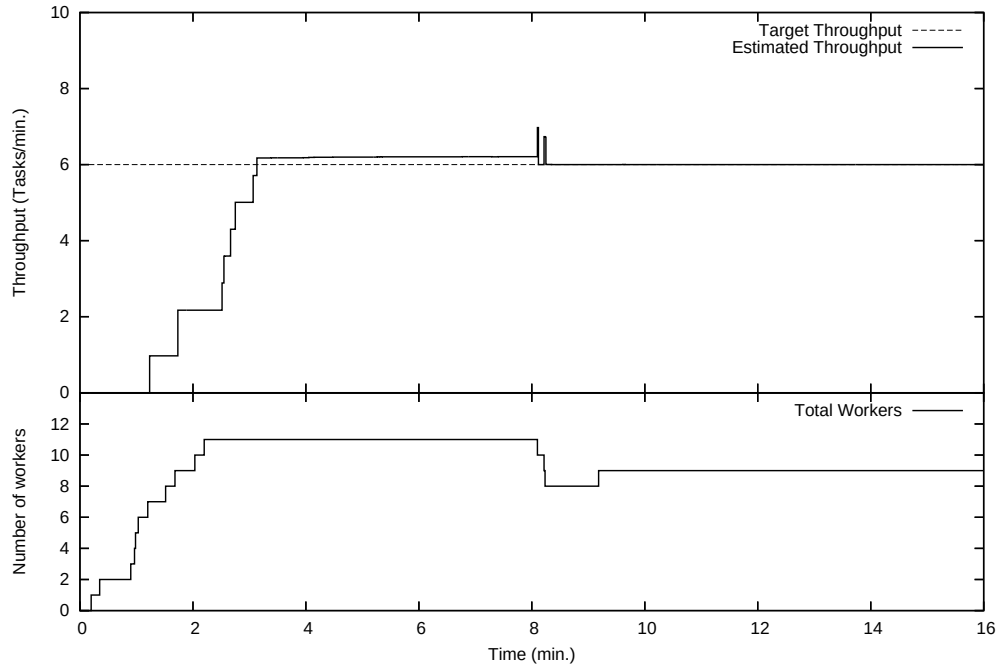


Figure 6: Performance adaption on a task farming application on a grid.

node on the skirit cluster (Brno, Czech Republic) and the glamdring (Plzen, Czech Republic) and cluster3 (Berlin, Germany) machines.

Since the nodes on the fs0 and skirit machine are reserved exclusively for the workers, and the cluster3 and glamdring machines where idle at the time of the experiment, the graph shown in Figure 6 is does not shown any performance fluctuations, even though the application is now running in a heterogeneous grid environment.

The workers on the skirit and glamdring machines are running faster than those on the fs0 (50 and 60 seconds per task, respectively). The worker on the cluster3 is significantly slower (130 seconds per task). Finally, 9 machines are required to reach the target performance. Due to the larger scheduling delays on the GridLab testbed, the number of workers does take somewhat longer to stabilize than in the previous experiment.

## 2.2 Transfer Protocol Optimization

### Scenario

This scenario deals with the migration of a parallel (e.g., Cactus) run from one machine to another machine. Consider that the application needs a large input data set that must be transfered from the source to the destination machine. The data should be transfered as efficiently as possible. GridFTP (and other transfer protocols) performance depends heavily on the characteristic of the network link, such as latency and available bandwidth, and on the transfer parameters that are used. Therefore, these parameters (e.g., send and receive buffer sizes, the number of parallel streams) should be carefully tuned for each separate link to achieve optimal performance.

To implement this use case, the following information is required:

1. amount of data to transfer
2. estimated network capacity between the sites
3. estimated network delay between the sites
4. maximum allowed send and receive buffer sizes on both sites

The amount of data to transfer must be provided by the user (or client application). The rest of the information is gathered by the Delphoi. Using the data described above, we can now compute the TCP stream buffer size as follows:

$$\text{bufsize} = min(\text{max-send-buffer}, \text{max-receive-buffer})$$

Using the product of network delay and capacity, we are able to determine the amount of data that the sender can write into the network before the first acknowledgment is received from the receiver. If the TCP buffer size can be set large enough to contain the total amount of data, a single TCP stream will suffice.

If the TCP buffer size cannot be set large enough, however, a single TCP stream will never be able to use the full capacity of the network, because the sender will not be able to write enough data into the buffer to keep the network filled. As a result, the sender will eventually block, waiting for an acknowledgment from the receiver.

By using multiple streams, the sender can continue sending data on a different TCP stream, even if previous streams are blocked. We can calculate how many TCP streams are required to fully use the capacity of the network as follows:

$$\text{streams} = 2 \times \left\lceil \frac{\text{delay.roundtrip} \times \text{bandwidth.capacity}}{\text{bufsize}} \right\rceil$$

As described in the previous deliverable, the implementation of this use case consist of a module in the Delphoi. This allows efficient access to the required low-level data. This module can be accessed using the *estimateTcpOptions* method. Using the *estimateNetworkMetric* method of the Delphoi, the low-level data itself can also be retrieved.

The Data Movement Service (developed by WP-8) uses the information provided by the Delphoi module to optimize data transfers.

### Results

To evaluate the transfer optimization component, we have used a benchmark that transfers a large amount of data between two sites in the GridLab testbed. To optimize this data transfer, the *estimateTcpOptions* method of Delphoi is used to retrieve a prediction for the optimal TCP settings. The exact amount of data transfered is adapted to the expected available bandwidth between the sites at the time of the test (as predicted by the *estimateNetworkMetric* method of Delphoi), and varies between 50 MB and 1 GB.

The benchmark then transfers the selected amount of data between the two sites using the predicted number of TCP streams. By comparing the predicted transfer time with the measured transfer time, we can evaluate the quality of Delphoi's predictions.

Two additional transfers are then performed, one using 50% more and one using 50% less TCP streams. The amount of data transferred is not changed. By comparing the transfer times obtained using the different number of streams, we can evaluate if our stream prediction was accurate, if using more streams would have yielded better results, or if using less streams would have been sufficient.

We have selected five sites in the GridLab testbed to run the benchmark at regular intervals. The sites and their location are shown in Table 1.

Table 1: Locations of the sites used in the experiment.

| Name | Location |
|---|---|
| fs0.das2.cs.vu.nl | Amsterdam, The Netherlands |
| skirit.ics.muni.cz | Brno, Czech Republic |
| cluster3.zib.de | Berlin, Germany |
| eltoro.pcz.pl | Czestochowa, Poland |
| helix.bcvc.lsu.edu | Louisiana , USA |

The benchmark transfers data between three combinations of these five sites. The combinations are shown in Table 2. The table also shows approximations[1] for the amount of data transferred, and the network delay, capacity and available bandwidth. The results of the experiments are shown in Figures 7, 8 and 9.

Table 2: Selected Sited for Transfer Time Optimization.

| Source | Destination | Latency (ms) | Capacity (MBit/s) | Available (MBit/s) | Size (MB) | Remark |
|---|---|---|---|---|---|---|
| fs0 | eltoro | 75 | 97 | 83 | 300 | Normal link |
| cluster3 | helix | 155 | 60 | 23 | 90 | High delay |
| fs0 | skirit | 19 | 800 | 610 | 1024 | High bandwidth |

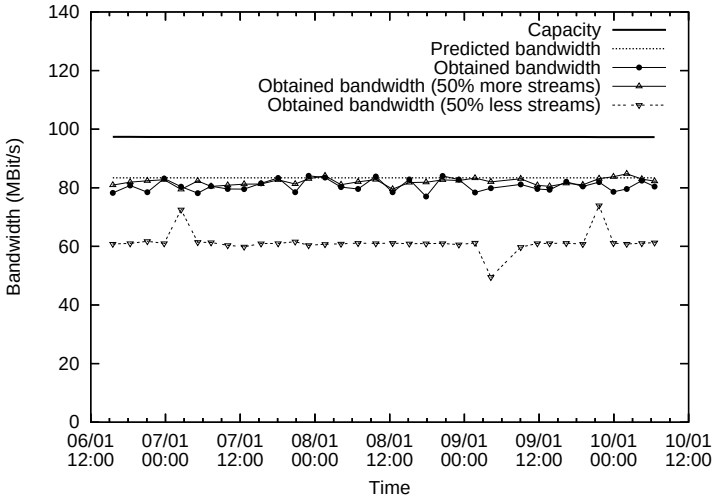


Figure 7: Transfer times from fs0.das2.cs.vu.nl to eltoro.pcz.pl.

[1]The actual values may vary over time.

Figure 7 shows the results of the transfer optimization test between the fs0 and eltoro machines. Besides showing the bandwidth obtained by the data transfers, it also shows the transfer time prediction (as predicted bandwidth) and the network capacity. Both are provided by Delphoi. As the figure shows, the bandwidth obtained using the predicted TCP settings, is very close the predicted bandwidth. Adding 50% more streams does not significantly improve the obtained bandwidth, while removing 50% of the streams does have a significant impact. Therefore, the TCP settings and transfer time predictions are correct.

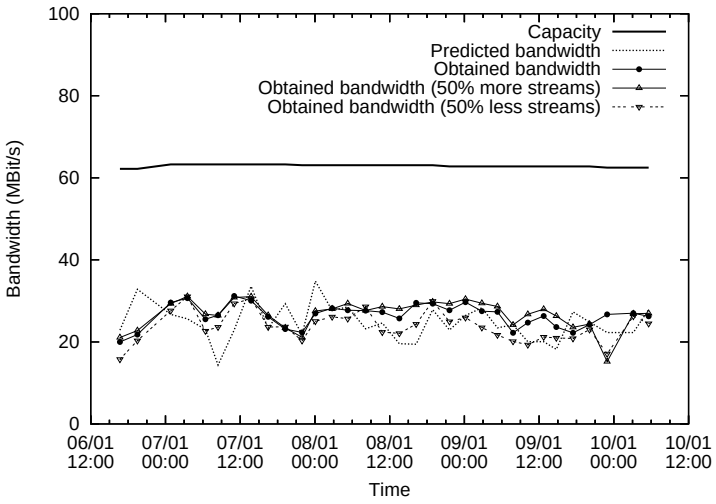


Figure 8: Transfer times from cluster3.zib.de to helix.bcvc.lsu.edu.

Figure 8 shows the results of the transfer optimization test between the cluster3 and helix machines. Again, the bandwidth obtained using the predicted TCP settings is very close the predicted bandwidth, and adding more streams does not improve the obtained bandwidth much. This time, however, the impact of removing streams is much smaller.

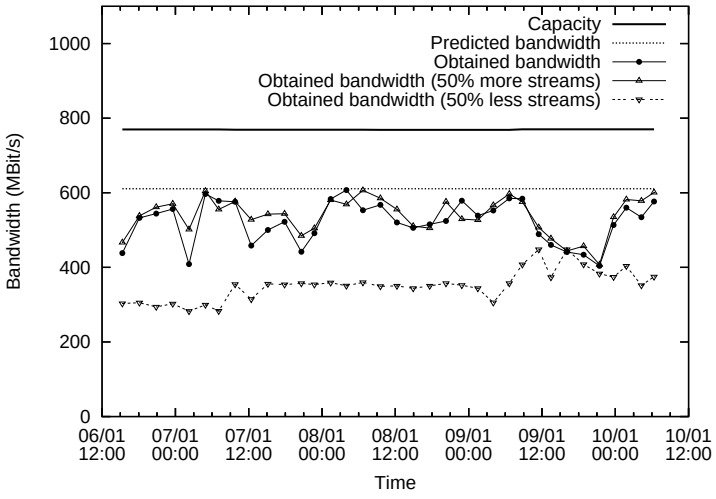


Figure 9: Transfer times from fs0.das2.cs.vu.nl to skirit.ics.muni.cz.

The results of our final experiment are shown in Figure 9. This figure shows the results of the

optimization test between the fs0 and skirit. Again, the predicted number of TCP streams is sufficient. Adding 50% more streams only slightly improves the obtained bandwidth, and the impact of removing half of the streams is high. Unlike in the previous experiments, however, the predicted bandwidth is not achieved. The network capacity between these two machine is very high (in the order of 800 Mbits/s), making it hard to fill the link, especially if there is a significant load on the machines. As a result, the benchmark is not capable of obtaining the predicted bandwidth of 610 Mbit/s, even if more streams are added.

These experiments show that the TCP option estimations produced by Delphoi can be used by applications to improve performance of the data transfer. In two out of three experiments the transfer time was correctly predicted. In all three experiments the transfer time showed only little change if more than the predicted number of TCP streams was used. Using less streams, however, severely degraded the performance in two of the three cases.

## 2.3 Replica Selection

### Scenario

The scenario is again about the migration of a parallel (e.g., Cactus) run from one machine to another machine. This time, consider that the application needs a large input data set that has been replicated several times throughout the grid. The time needed to transfer the application input data from a replica to the target machine needs to be estimated in advance, for possibly multiple purposes. First, a migration target machine may be selected by the time it takes to migrate to it (in behalf of the GRMS resource broker, as developed by WP9). Second, after a migration target has been chosen, one of possibly many replicas must be selected. Third, after a target machine and source replica have been selected, the transfer software/protocol may be optimized (like GridFTP parameter tuning). Finally, an estimation of data transfer time may be necessary to start migration in time, before the allocated compute time on the migration source machine ends.

To implement this use case, the following information is required:

1. amount of data to transfer (provided by the user).
2. estimated available network bandwidth from each of the sites to the target site.

Using the amount of data to transfer and the estimate of the available network bandwidth between the sites, the transfer time per pair of sites can be estimated as follows:

$$\text{transfer time} = \frac{\text{bandwidth.available}}{\text{data size}}$$

The site with the lowest transfer time to the target site can then be selected.

The implementation of this use case consists of a module in the Delphoi. This allows efficient access to the required low-level data. This module can be accessed using the *estimateTcpOptions* method.

The information provided by the replica selection module of the Delphoi is used by the *Replica Catalog* developed by WP-8.

### Results

To evaluate this use case, we have implemented a benchmark which uses the same replica selection strategy as the Replica Catalog. This benchmark sends a request to the Delphoi to predict the transfer times of 100 MB of data from a set of machines to a single target machine. The data is then transferred from each of the machines (using the benchmark developed for the previous use case), and the actual transfer time is compared to the prediction. We use the same set of GridLab machines as in the previous use case (shown in Table 1).

The results of the first experiment are shown in Table 3. There, the data needs to be replicated to the fs0 machine from one of the other four machines.

Table 3: Replicating 100MB of data to the fs0.

| Source | Predicted Time (sec.) | Measured Time (sec.) |
|---|---|---|
| skirit | 2.7 | 1.7 |
| cluster3 | 8.5 | 8.9 |
| eltoro | 9.7 | 11.5 |
| helix | 13.9 | 60.1 |

As the table shows, the predicted transfer time is smallest for the skirit machine. Therefore, the replica catalogue would choose this machine as the source for the data transfer to the fs0. The measurements show that this is indeed the right choice.

Table 4: Replicating 100MB of data to the cluster3.

| Source | Predicted Time (sec.) | Measured Time (sec.) |
|---|---|---|
| skirit | 8.8 | 8.9 |
| fs0 | 9.2 | 8.9 |
| eltoro | 10.3 | 11.5 |
| helix | 16.1 | 34.7 |

For the second experiment, the data needs to be replicated to the cluster3 machine. The results are shown in Table 4. Like in the previous experiment, the predicted transfer time is smallest for the skirit machine. The replica catalogue would choose the skirit as the source, and the measurements show that this is indeed the right choice.

## 2.4 Remote Data Visualization

### Scenario

A user wants to visualize the output of a completed, or even a running, remote computation (like a Cactus run). The critical resource is the network bandwidth that limits either the quality of the image, or the delay of the visualization, or even both.

The user needs to determine a desired trade-off between quality and delay. For this purpose, the user specifies the minimal useful resolution of the image, and a desired frame rate. The remote visualization software can then provide the best possible images within the constraints of the user and the technical possibilities.

**Design**

The key to this adaptation problem is to perform visualization using progressive resolution. The visualization software can construct the image hierarchically, starting with a very low resolution, up to the maximum quality that can be achieved within the constraints of frame rate and network bandwidth. The resolution may even vary between different parts of an image.

The adaptation component uses its network bandwidth data from the Delphoi system, produces short-term predictions, and correlates this with transfer times of the needed data over the measured link. The adaptation component then provides an estimate for the data volume that can be sent within a given time limit.

An adaptive component in the visualization system (Hierarchy Manager) links the adaptation information with user defined visualization parameters. These parameters are:

1. maximal acceptable wait time for initial image, and
2. minimal usable resolution,
3. frame rate.

**Data Hierarchy:** A very common approach to allow for visualization with adaptive resolution is to subdivide the image or data space into octree nodes. The root node of the octree (on level 0) has the coarsest resolution, the nodes on the higher levels $(1, 2, ...)$ have each a doubled resolution.

The root node is transferred first, then the nodes on level 1 are transferred in some order, then the nodes on level 2 and so on. Upon data arrival, the blocks are each processed and displayed independently. That procedure results in a visualization of increasing resolution, with a maximal data transfer overhead of 12.5% (3D, doubled resolution on each level).

If a new frame is requested, the blocks not yet received are abandoned, and a new root node for the next frame data is requested.

**Case a: The user gives a time limit for the first image to appear** The bandwidth and transfer time estimate is used to determine the maximal possible data and image resolution within the given time limit (initial resolution). The octree hierarchy is created to reflect that resolution for the root node.

**Case b: The user gives a limit for the minimal usable resolution** The octree hierarchy is created to directly reflect that resolution for the root node.

**Case c: The user sets a fixed frame rate** The bandwidth and transfer time estimate is used to determine the maximal transferable octree depth within the given time limit. The Octree blocks are retrieved as described, with minimal delay for the root node, and providing the defined frame rate in the maximal possible resolution. If full resolution cannot be achieved at the given frame rate, only lower levels of the octree are used.

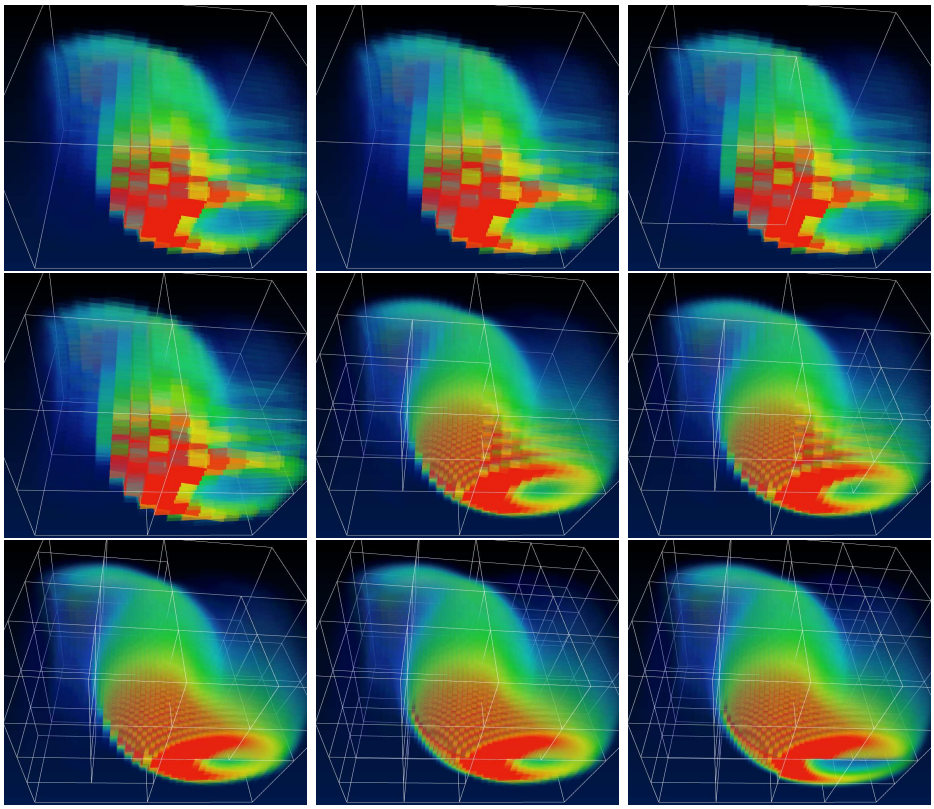

Figure 10: The sequence depicts the volume rendering of a remote data set. First, a coarse resolution representation of the data is generated on-the-fly and transferred to the local visualization client. Next, octree nodes closer to the point-of-interest (in this case, the camera position) are requested and integrated at progressively higher resolutions.

**Case d: The user sets a combination of parameters** The bandwidth and transfer time estimate is used to determine the maximal possible data and image resolution within the given time frame (initial resolution). If the known minimal resolution cannot be reached in that given time frame, the QoS contract is broken, and the data for the minimal resolution are transfered. Otherwise, the data for the initial resolution are transfered. If the given frame rate allows additional data transfers, the image is consecutively refined with additional incoming data.

### Implementation

The implementation of the bandwidth and transfer time estimation are described in report GridLab-7-COMP-0004-1.0.

The client library used to obtain the data is available at
*http://www.gridlab.org/Internal/WorkPackages/wp-7/software.html*.

The implementation of the remote data access infrastructure we have described was based on an experimental version of the GridFTP server provided by the Globus Group. This server was not part of the Globus software distribution at that time. Any implementation of a GridFTP-Service following the GGF GridFTP Specification 1.0 (hence implementing ESTO/ERET support) or newer should (with the appropriate extension) be able to be used as service host.

The Octree setup uses short-term prediction obtained by linear regression from the obtained estimation data. That was implemented in a module to the Amira visualization system named `HxHierarchyManager`. That module controls the Octree setup, and the invocation of the remote data reader module (`HxHDF5External`).

The Amira modules are also available at
*http://www.gridlab.org/Internal/WorkPackages/wp-7/software.html*.

**Results**

For benchmarking the software we used a dual Xeon 1.7GHz Server running RedHat Linux 8.0 as a data server. The machine was equipped with 1GB of RAM and a logical volume storage of 320 GByte (36.5 MByte/sec transfer rate). The measurements have a granularity of 1 second.

The renderings have been performed on a dual Pentium IV system with 2.6 GHz, 1 GByte main memory and NVidia Quadro4 graphics. The system ran under RedHat Linux 8.0 with the standard NVidia video driver.

In order to evaluate our approach, we performed a number of performance measurements for accessing, loading and displaying large remote HDF5 data sets. We compare the performance obtained using the GridFTP plugin (*GridFTP HDF5*) with a comparable remote access technique, that is HDF5 over GridFTP partial file access (*GridFTP PFA*). We also include measurements of local (*local access*) and Network File System (*NFS access*) times to see if we achieved our goal of having acceptable waiting times before the first visualization is created, considering the local and NFS times as acceptable.

The results of these tests are listed in Table 5. The time needed to create the first image ($t_3$) is composed of the time needed to gather and transfer the meta data ($t_1$) and the time needed to filter and transfer the subsampled first timestep ($t_2$). $t_4$ gives the access time for a full resolution time step.

The tests have been performed on a Local Area Network (*LAN*) with normal network load (latency 1ms, measured 32.0 MBit/sec), and on a Wide Area Network connection (*WAN*) between Amsterdam and Berlin (latency 20ms, measured bandwith: 24.0 MBit/sec).

The *WAN* measurements have been performed with various *level* settings, that is with different depth of the octree hierarchy created.

These measurements show that the goal of a fast initial visual representation of the data set was achieved: a small startup time $t_3$ can be achieved by using the *GridFTP HDF5* technique combined with hierarchical access ($level \geq 2$). This time is of the same order of magnitude as for local visualization.

Specifying the hierarchy level provides the user with an interactive mechanism for tuning response times. The data access scheme could prove its adaptivity for different network connectivity. In principle, the user can reduce the time to obtain a first visual representation by choosing a larger hierarchy level. The tradeoff for shorter startup times is the total transfer time for a fully resolved data set (all octree levels)[2]. The results show that relation ($t_3$ / $t_4$) clearly for the WAN measurements with different level settings.

Also, the large overhead for the complicated meta data access was dramatically reduced in comparison to GridFTP partial file access. The remaining time difference relative to the NFS

[2]The maximum amount of additionally transferred data caused by the octree based access scheme is on the order of 15%. The higher number of resulting block requests increases the *overall* transfer time also due to the additional latencies.

| Access Type | Net | Level | Meta Data $t_1$ | Root Block $t_2$ | Startup $t_3 = t_1 + t_2$ | Complete $t_4$ |
|---|---|---|---|---|---|---|
| local access | - | 2 | 7 sec | 1 sec | 8 sec | 3 sec |
| NFS access | LAN | 2 | 8 sec | 5 sec | 13 sec | 8 sec |
| GridFTP HDF5 | LAN | 2 | 11 sec | 2 sec | 13 sec | 11 sec |
| GridFTP PFA | LAN | 2 | 165 sec | 10 sec | 175 sec | 200 sec |
| GridFTP HDF5 | WAN | 3 | 14 sec | 2 sec | 16 sec | 126 sec |
| GridFTP HDF5 | WAN | 2 | 14 sec | 3 sec | 17 sec | 68 sec |
| GridFTP HDF5 | WAN | 1 | 14 sec | 7 sec | 21 sec | 45 sec |
| GridFTP HDF5 | WAN | 0 | 14 sec | 41 sec | 55 sec | 41 sec |
| GridFTP PFA | WAN | 3 | 430 sec | 28 sec | 458 sec | 3760 sec |
| GridFTP PFA | WAN | 2 | 430 sec | 53 sec | 483 sec | 960 sec |
| GridFTP PFA | WAN | 1 | 430 sec | 110 sec | 560 sec | 477 sec |
| GridFTP PFA | WAN | 0 | 430 sec | 220 sec | 670 sec | 220 sec |

Table 5: The table lists performance measurements for the various access techniques we explored. The results have been obtained by timing the visualization process for a 32 GB HDF5 file, containing 500 timesteps, each timestep with the resolution of $256^3$ data points (double precision).

meta data access results from the application of the zero filter to all data sets, the time needed to write the meta data file, and the time to transfer it.

To summarize, while using automatic adaptation, the user, as expected, can influence the individual values for '$t_2$' and '$t_3$' by specifying time and resolution constraints, without the need to know the details of the transfer algorithm, or the need to adjust the complex parameter space. That was the declared goal of the use case implementation.

**Acknowledgements**

The work on this use case was an collaborative effort between VU Amsterdam (WP-7) and ZIB Berlin (WP-8). The work described above was published in [HMK+03]. That publication describes the use case in much more detail, and also includes detailed time measurements.

## 2.5 Queue Waiting Time Estimation

### Scenario

The GRMS resource broker, as developed by WP-9, is supposed to schedule an application request such that the application starts running as soon as possible.

For this purpose, the scheduler needs information about predicted wait time in processor queues when scheduled to a given machine, or a combination of multiple machines, the latter in case of co-allocation.

Our current implementation is designed to predict waiting times in batch queuing systems. An example of such a system PBS, which is widely used throughout the GridLab testbed.

For this use case the state of the queue(s) and/or jobs on a resource must be frequently monitored (e.g., once a minute). For each job in a queue, the following information is recorded:

1. submission time of the job
2. state of the job (e.g., waiting or running)
3. start time of the job
4. number of processors required by the job (optional).

By calculating the total number of processors used by all running processes, the utilization of the queue can be determined, provided that the total number of processors available to that queue is known.

Next, for all waiting jobs, the waiting time so far is determined by subtracting the current time from their submission time. This information is stored for future reference.

In general, the waiting time of a job may depend on its size (i.e., the number of hosts required to run it). Large jobs, for example, must often wait longer for the required resources to become available. Therefore, the job size is used in the waiting time prediction (if available). Because it is difficult to treat jobs of each possible number of hosts separately, 4 classes of job sizes are currently used, *single* (1 host), *small* (2 to 4 hosts), *medium* (5 to 16 hosts), and *large* (17 or more hosts).

For each class of jobs, the current maximum and average waiting times can then be determined. Using these values, and the forecaster library of the Delphoi system, the waiting time of new jobs can be predicted.

The implementation of this use case consists of a queue monitoring module in the Delphoi system. This module frequently retrieves information about the state of the queues on the system, and stores this information for future reference. The information queue waiting time can be retrieved from the Delphoi using the *getQueueWaitingTime* call.

**Results**

To evaluate the queue waiting time estimation, we do not use the Delphoi directly. Instead, we use a simulation consisting of the core of the queue information module used by the Delphoi. We then simulate queue traffic by replaying log files from the PBS queuing system (used on several machines in the GridLab testbed). This allows us to perform experiments and evaluate the behavior of our system over longer periods of time.

Our simulation traverses through the PBS log file and simulates the job queue. Whenever the log file states that a job was submitted, a Job object is created, containing the submission time of the job, and a prediction of how long this job must wait. The Job object is then stored until the log file states that a job was started. At that time, the real waiting time of the job is known and the prediction error can be determined.

To predict how long a job must wait, the core of the queue information module of the Delphoi system is used. This module retrieves queue information once every (simulated) minute. For every waiting job, it records how long the job has been waiting and, if available, how many machines it requires. This information can then be used to estimate the waiting times of future jobs.

Figure 11 shows the result for a PBS log of the fs0.das2.cs.vu.nl[3], covering the period of 1 to 26 March 2004. Because this machine is mainly used for research into parallel and distributed computing, most jobs have short lifetime and the number of jobs is high. March 2004 was a particularly busy period: a total of 91349 jobs where processed by the queuing system (an average of 146 jobs an hour).

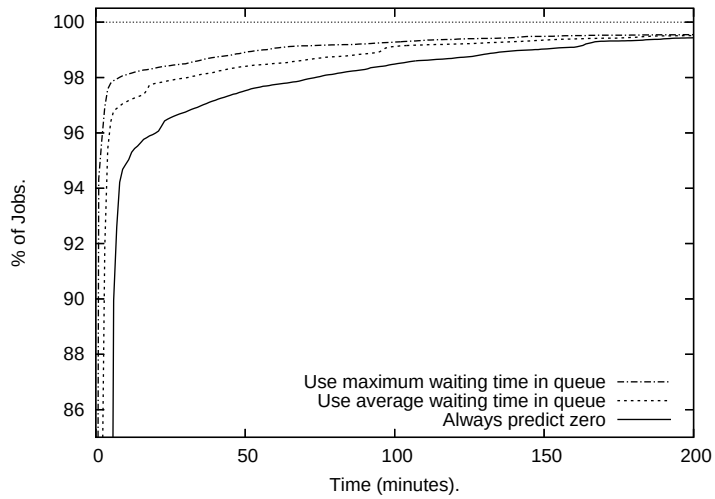


Figure 11: Error in Job waiting time estimation on fs0 (log of march 2004).

Figure 11 shows the results for three different methods of estimating the job waiting time. The first approach simply assumes that jobs will never wait, and always predicts a waiting time of zero. The second approach calculates the average waiting time of the jobs currently in the queue, and uses this value as an estimation for the waiting time of new jobs. The third approach uses the maximum waiting time of the jobs currently in the queue.

Figure 11 shows the percentage of jobs for the error in the waiting time estimation is less than a certain amount of time. For example, when using zero as an waiting time estimation, 68% of the jobs have an estimation error of 5 minutes or less and 98% of the jobs are run within 75 minutes of this estimation. If the average is used as an estimation, however, these number are improved considerably: 96% of the jobs have an error of 5 minutes or less and 98% of the jobs is run within 31 minutes of their estimated time. Using the maximum waiting time yields the best results: almost 98% of the jobs is run within 5 minutes of the estimated time.

Figure 12 shows the result for a PBS log of a queue on the skirit.ics.muni.cz, covering the period of 13 March to 13 April 2004. Because this machine is mainly used for running production jobs, waiting times can be high. The number of jobs processed is low compared to the fs0: only 1826 jobs where processed in the entire period.

The long waiting times have a significant impact on the accuracy of the waiting time estimation. When using zero as an waiting time estimation, 65% of the jobs have an estimation error of 5 minutes or less and only 80% of the jobs are run within five hours of this estimation. Using the average results in a significant improvement: 79% of the jobs have an error of 5 minutes or less and 87% has an error of five hours or less. Like in the previous figure, using the maximum

[3]Unfortunately, the PBS logs recorded on the fs0.das2.cs.vu.nl do not contain information about the size of the jobs. They only state when jobs are queued, started, and finished. We therefore assume all jobs to be of equal size.

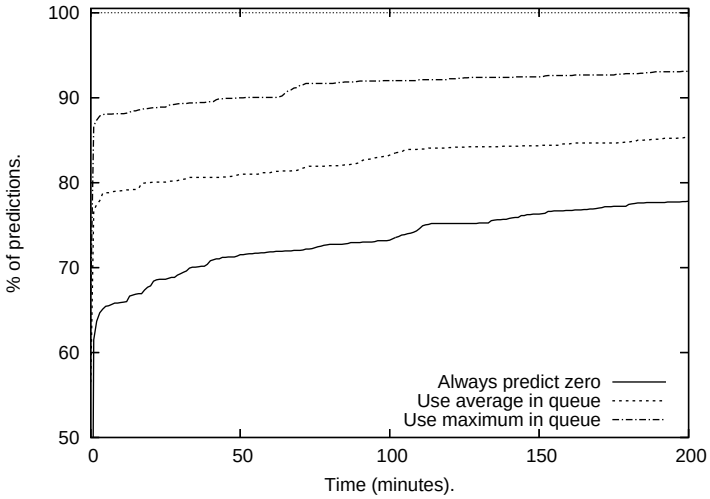


Figure 12: Error in Job waiting time estimation on skirit (from 13 March'04 to 13 April'04).

waiting time as an estimation yields the best result: 88% of the jobs have an error of 5 minutes or less and 94% is run within five hours of the predicted time.

The results shown in Figures 11 and 12 indicate that the queue information gathered by Delphoi can be used to provide good job waiting time predictions to GRMS. Although there is clearly room for improvement, using simple estimation schemes such as taking the average or maximum current waiting time already produces reasonable results. The flexible and extensible architecture of Delphoi makes it an excellent platform for further research into this area.

# 3 Summary

In this report, we have evaluated the adaptive components developed within GridLab's work package 7. Guided by the use cases outlined in GridLab-7-UCR-0001-2.0, we have presented five scenarios for which our adaptive components have been built and integrated with other services developed by GridLab. For each use case, we have briefly outlined the implementation approach for the adaptive component, and have described the performed evaluations, and the achieved results. For all use cases we were able to show that our adaptive components successfully improve the performance of their client grid services. As our Delphoi service, accompanied by the adaptive components, is permanently deployed on GridLab's testbed, we were able to demonstrate its suitability for dynamic performance adaptation in grid environments.

# References

[HMK+03] Andrei Hutanu, Andre Merzky, Ralf Kähler, Hans-Christian Hege, Brygg Ullmer, Thomas Radke, and Ed Seidel. Progressive Retrieval and Hierarchical Visualization of Large Remote Data. In *Proceedings of the 2003 Workshop on Adaptive Grid Middleware*, pages 60–72, September 2003.