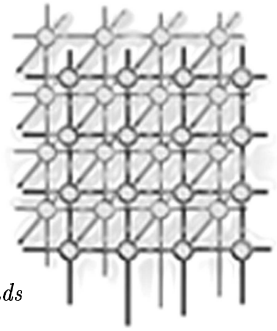
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **1**:1 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Middleware Adaptation with the Delphoi Service



Jason Maassen, Rob V. van Nieuwpoort,
Thilo Kielmann, Kees Verstoep, Mathijs den Burger

Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{jason,rob,kielmann,versto,mathijs}@cs.vu.nl
http://www.gridlab.org/delphoi/

SUMMARY

Grid middleware needs to adapt to changing resources for a large variety of operations. Currently, however, there is only low-level information available about Grid resources, coming from various, but functionally isolated monitoring and information systems. In this paper, we present the Delphoi service. It both provides a unified interface to the necessary information, and also matches the abstraction level needed by middleware services to adapt their behavior. We describe Delphoi's architecture, the information it provides, and we evaluate the quality of its performance information. Delphoi has been developed as part of the EC-funded GridLab project and is currently being deployed on the project's testbed for adding adaptivity to GridLab's middleware services.

KEY WORDS: resource monitoring, prediction, network performance, file transfer, queue waiting time, event logging

1. Introduction

Grids can offer unprecendented opportunities to resource-hungry applications. However, availability and performance of Grid resources (processors, network links) are highly variable. Traditional application codes, oblivious of this resource variability, hardly achieve acceptable levels of performance.

In order to adapt application behavior to actual resource performance, Grid middleware is needed both to provide information about the status of the resources and to guide decisions about adapting application performance. Unfortunately, the many pieces of information needed for this purpose are typically scattered among many sources, and are low-level, resource-centric data items. In consequence, adaptive middleware is rather complicated and much functionality gets re-implemented by each adaptive middleware component.

In this paper, we present the Delphoi service that provides unified access to various information sources. Delphoi has been developed by the EC-funded GridLab project in order to provide adaptivity for a variety of use cases of the Grid Application Toolkit (GAT). Besides

Received 17-12-2004

access to information, Delphoi provides translation to high-level information in combination with prediction* into the near-term future. Delphoi has a flexible and extensible system architecture that also supports application-specific metrics and event logging. Delphoi is currently being deployed in GridLab's testbed for a variety of performance optimization use cases.

The remainder of this paper is organized as follows. Section 2 outlines the need for behavior adaptation by GridLab's Grid Application Toolkit (GAT). Section 3 presents the Delphoi service emphasizing its high-level interface to resource-related data. Section 4 presents Delphoi's architecture and distributed implementation. In Sections 5 and 6 we evaluate the information provided by Delphoi using several use cases, among which are the fully automated adaptation of data transfers and the estimation of job waiting times in batch queueing systems. We discuss related work in Section 7 before we conclude in Section 8.

2. Adaptation in the Grid Application Toolkit

The GridLab project [1] aims at supporting application development for Grids. The main product is the Grid Application Toolkit (GAT) [2]. The GAT's main objective is to provide a single, easy-to-use Grid API, while hiding the complexity and diversity of the actual Grid resources and their middleware layers.

Figure 1 shows the GAT software architecture. It mainly distinguishes between user space and capability space. The application code, that has been programmed using the GAT API, is running in user space. The GAT *engine* is a lightweight layer that dispatches GAT API calls to service invocations via GAT *adaptors*. Adaptors are specific to given services and hide all service-specific details from the GAT. A GAT engine typically loads adaptors dynamically at runtime, whenever a certain service is needed. We currently have GAT implementations and language bindings for C, C++, Python, and Java.

While application and GAT together run in user space, the services are executed in the so-called *capability space*, which is distributed across the machines of a virtual organization (VO). The capability space comprises the resources themselves (hosts, data, etc.) and the middleware providing services to access them. The GAT adaptors can directly access standard Grid middleware like Globus [15] or Unicore [8]. For more advanced functionality, GridLab also provides higher-level services like the GRMS resource broker [16] or the services for remote file access, data movement, and replication management [20]. Within the GridLab software framework, these services are the ones having the most demand for behavior adaption based on current and expected resource availability and performance. In particular, we are dealing with the following adaptation use cases:

- Data transfer protocol optimization.
  For job execution on a Grid node, data and program files need to be transferred to

*The ability to answer questions like "How long will my job have to wait?" made us name the service after the Delphoi oracle, known from ancient Greek mythology.

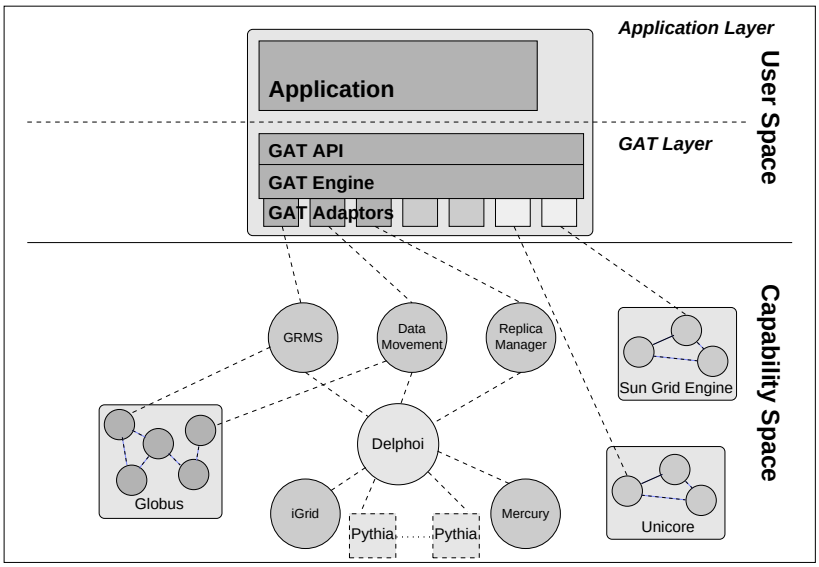


Figure 1. GridLab's GAT framework software architecture.

the execution site before program start. Resulting data sets need to be transferred back afterwards. As these data sets can be quite large, it is important to optimize data transfer times, depending both on the available software (like GridFTP [14]) and on the network performance characteristics [12].

- Replica selection.
  Whenever more than one replica of a given file is available, it is beneficial to select the one that can be transferred with the shortest completion time to a given site of a VO. Assuming data transfer protocol optimization will take place as in the previous use case, it also becomes necessary to estimate the transfer times for each replica.
- Remote data visualization.
  Some data sets may become so large (e.g., in the order of terabytes) that making a local copy becomes infeasible. Instead, such files need to be accessed directly as remote files. For example, for visualizing large data sets generated by a Cactus [3] simulation, remote data access needs to be adapted to user constraints (waiting time, image quality) and network capacities.
- Job waiting time estimation.
  The GRMS resource broker is used for scheduling jobs on machines of a VO, trying to minimize the job completion time. Part of this problem is finding machines on which jobs will be started as soon as possible. For this purpose, GRMS needs performance-related information about machines, the available queues of their local scheduler, and the currently expected waiting time for a given job in these queues.

These use cases have in common that they need various kinds of data about the resources within a VO, in order to combine this data into higher-level performance information. Typically, the required data is scattered among various sources. Lookup and retrieval of this data, along with the presentation of those as high-level information are frequently recurring tasks for Grid middleware. Our Delphoi service implements these tasks for higher-level Grid middleware services in need of performance adaptation.

3. The Delphoi Service

In this section we explain the functionality provided by the Delphoi service. Whenever possible, we give examples how these pieces of functionality are used by GridLab components. We will show several calls of the Delphoi interface, and explain how they can be used to retrieve information. These interfaces are general, and can be implemented in many different ways. As we will describe in Section 4, we currently provide socket and Web Service implementations of this interface.

3.1. Meta-information Methods

Delphoi provides several methods that can be used to get meta-information about what data are collected on which resources. For instance, the following method returns a list of hosts that Delphoi has information about:

```
String[] getActiveSites()
```

Another method returns a list of the metrics that are collected for a given resource:

```
MetricDescription[] knownMetrics(String hostName)
```

The *MetricDescription* object contains information about a particular metric, such as its name, parameters and the result that will be returned. For example, the *MetricDescription* for the *queue.waitingtime* metric is shown in Table I.

This MetricDescription shows that six parameters are required to retrieve the *queue.waitingtime* metric; a hostname, jobmanager name, queue name, job size, and a time frame consisting of a start and end time. This time frame can be in the past, the future, or it could start in the past and end in the future. In the latter two cases, Delphoi will return predictions based on historic data. If the start and end times are not provided (i.e., *null*), the most recent data available will be returned.

A MetricDescription also describes the metric value that is returned. As Table I shows, a *queue.waitingtime* metric consist of three *Float* values containing the average number of waiting jobs, average waiting time of these jobs, and standard deviation of the waiting times.

3.2. Generic Information Retrieval

Using the meta-information retrieved using the aforementioned calls, information on a specific resource can be extracted from Delphoi. Information can be from the past, present or future. In the latter case, Delphoi will try to return predictions based on historic data. Any metric value can be retrieved using the following call:

Table I. Metric description for the tcpoptions metric.

Generic Fields	
Name	"queue.waitingtime"
Comment	"Waiting time information for a queue"

Parameters	
String	Source hostname
String	Jobmanager
String	Queue name
Integer	Job size
Calendar	Start of time interval
Calendar	End of time interval

Results	
Float	Average waiting jobs
Float	Average waiting time
Float	Standard deviation waiting time

```
Map getMetric(String metric, Map parameters);
```

The *metric* parameter contains the name of the metric to be retrieved, while *parameters* map should contain the necessary parameter values for this metric. The returned map will then contain the result values, as described in the MetricDescription.

For convenience, the Delphoi also offers several methods which are specialized in retrieving a certain type of information. Their implementation simply translates the call to an appropriate invocation of *getMetric*. These convenience methods will be described in the following sections.

3.3. Low-level Network Information

To retrieve low-level network performance data, such as bandwidth, capacity or latency, the following call can be used:

```
DoubleMeasurement[] estimateNetworkMetric(
  String sourceHostName,
  String destinationHostName,
  String metric,
  String operation,
  Calendar startTime,
  Calendar endTime)
```

The hostnames of both the source and destination machines must be specified as parameters. The "metric" parameter specifies which network metric must be retrieved. A list of network metrics that are currently supported (in accordance with [12]), is shown in Table II.

The *operation* parameter can be used to apply an operation to the retrieved measurement data. Currently the "min", "mean", "max" and "all" operations are supported, indicating that the minimum, mean, maximum or all values in the given time frame should be returned.

Table II. Currently supported network metrics.

Metric	Description
path.delay.oneway	one way delay between two sites
path.delay.roundtrip	round trip delay between two sites
path.bandwidth.available	available bandwidth between two sites
path.bandwidth.utilized	used bandwidth between two sites
path.bandwidth.capacity	network capacity between two sites
path.topology	hoplist of network between two sites

The "startTime" and "endTime" parameters specify the time frame that the user is interested in. The result of the "estimateNetworkMetric" method is an array containing one or more "DoubleMeasurement" values (depending on the operation), each containing a timestamp and a value.

Delphoi also provides methods for retrieving multiple network metrics at a time, designed to improve the performance of lookups:

```
DoubleMeasurement[][] estimateNetworkMetricOneToMany(
  String sourceHostNames,
  String [] destinationHostName,
  String metric,
  String operation,
  Calendar startTime,
  Calendar endTime)

DoubleMeasurement[][] estimateNetworkMetricManyToOne(
  String [] sourceHostNames,
  String destinationHostName,
  String metric,
  String operation,
  Calendar startTime,
  Calendar endTime)

public DoubleMeasurement[][][] estimateNetworkMetricMatrix(
  String [] hostNames,
  String metric,
  String operation,
  Calendar startTime,
  Calendar endTime)
```

These methods are similar to the previous one, but can be used to simultaneously retrieve a single network metric for a *set* of hosts, thereby reducing the amount of communication required between a client and Delphoi. In addition, all queries which require additional network communication (e.g., information lookups in remote databases) can then be performed by Delphoi in parallel. This can reduce the reply time significantly.

3.4. High-level Network Information

In addition to low-level network metrics, Delphoi provides additional convenience methods that apply some intelligence to transform low-level metrics to useful application-level information.

```
public TcpOptions estimateTcpOptions(
    String sourceHostName,
    String destinationHostName,
    long dataSize,
    String transferMethod,
    int maxTcpStreams,
    Calendar startTime)
```

This call will give an estimation of the optimal configuration of a TCP connection between two machines. Given the source, destination and the number of bytes that has to be transferred, it predicts the TCP send buffer size and the number of parallel TCP streams (as proposed by [19]) that should be used for maximal performance. The "transferMethod" input parameter indicates the transfer method that will be used (e.g. GridFTP [14]). This can be any user-defined string; the purpose of this string will be explained below. The "maxTcpStreams" parameter can be used to limit the amount of parallel TCP streams that is allowed. The result of the method is a *TcpOptions* object that contains the prediction of the optimal number of parallel streams and the send and receive buffer sizes. In Section 5, we will show a scenario that uses this call to optimize data transfers.

```
public double estimateTransferTime(
    String sourceHostName,
    String destinationHostName,
    long dataSize,
    String transferMethod,
    Calendar startTime)
```

This call estimates the time it will take to transfer a specified amount of data from one grid site to another.

```
public double[] estimateTransferTimeOneToMany(
    boolean substractLoggedTraffic,
    String sourceHostName,
    String[] destinationHostNames,
    long dataSize,
    String transferMethod,
    Calendar startTime)

public double[] estimateTransferTimeManyToOne(
    String[] sourceHostNames,
    String destinationHostName,
    long dataSize,
    String transferMethod,
    Calendar startTime)

public double[][] estimateTransferTimeManyToMany(
    String[] hostNames,
    long dataSize,
    String transferMethod,
    Calendar startTime)
```

Like the previous call, these calls estimates the time it will take to transfer an amount of data from one grid site to another. For efficiency reasons, however, these calls can predict the time from one source to multiple destinations, from multiple sources to one destination, or from all sites in a list to all other sites in that list.

The call using multiple destinations is useful to a scheduler that must select a grid site for a job with a large input set or checkpoint file. The scheduler can invoke this method providing all possible candidate sites that can run the job, and then select the site with the smallest transfer time.

Within GridLab, the call using multiple sources is used for replica selection. For instance, if a file is replicated across several sites, and the "best", or "closest" replica must be chosen, this call can be used to find out what the transfer times of each replica to the destination would be. Next, the site with the smallest transfer time can be chosen.

```
public void logDataTransfer(
    String source,
    String destination,
    long dataSize,
    String transferMethod,
    TcpOptions options,
    Calendar startTime,
    Calendar endTime)
```

Using this method, applications can provide feedback to Delphoi about data transfers that have taken place. This feedback can be used by Delphoi to improve future predictions of data transfer times and TCP options. The method is optional, and is only used to improve predictions.

The "transferMethod" parameter can be any user defined string that indicates the transfer method used (e.g., "GridFTP"). By matching this parameter to the "transferMethod" parameters passed in the estimation method described above, Delphoi is able to differentiate between different transfer mechanisms. By correlating feedback to predictions with the same transfer method identifier, Delphoi can try to improve future predictions of each transfer method separately. The parameter "options" describes the TCP options that were used for the transfer, if applicable.

3.5. Queuing Information

To discover which batch queues are monitored, Delphoi provides the following call:

```
Queue[] getQueues()
```

getQueues returns a list of all queues which are currently being monitored. Each Queue object in this list contains the host name, the name of the job manager, and the queue name of a monitored queue. Multiple job managers may be available on the same host (e.g., PBS, Condor, LSF), and multiple queues may be available for one job manager.

These Queue objects can be used to retrieve more detailed information about a specific queue. For this purpose, the following calls are available:

```
QueueConfiguration getQueueConfiguration(Queue queue)
```

GetQueueConfiguration returns information about a queue's configuration. This configuration includes the number of hosts available to the queue, the number of CPUs in each host, and information on any limits that the queue imposes on use of CPU time, wall time, memory usage, or number of jobs. Although changes to a queue's configuration are infrequent, they may still occur occasionally. For instance, some hosts in a cluster could be down temporarily, decreasing the hosts available to the queue. Therefore, applications should not assume this information is static, but instead should contact Delphoi to retrieve it.

```
QueueWaitingTime getQueueWaitingTime(
    Queue queue,
    int jobSize,
    Calendar startTime,
    Calendar endTime)
```

GetQueueWaitingTime retrieves information about the waiting time of jobs in a queue in a specified time interval. It returns an object containing the average number of waiting jobs, average job waiting time, and the standard deviation of job waiting time in that interval. 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, getQueueWaitingTime requires the job size to be passed as a parameter. The information returned by getQueueWaitingTime will only be valid for jobs of the specified size. Because it is difficult to treat jobs of each possible number of hosts separately, Delphoi currently uses 4 classes of job sizes, *single* (1 host), *small* (2 to 4 hosts), *medium* (5 to 16 hosts), and *large* (17 or more hosts). Delphoi automatically matches queries to the appropriate job class.

```
ResourceUtilization getResourceUtilization(
    Queue queue,
    Calendar startTime,
    Calendar endTime)
```

GetResourceUtilization returns information about the average number of free hosts available to a queue in the specified time interval. This is a measure for the utilization of the machine.

3.6. Logging

Delphoi also provides an interface to retrieve logging information. Services or applications write log entries locally using Mercury[11]. Each log entry contains the following fields:

- *service:* name of the service which produced the log;
- *component:* name of the component within the service;
- *origin:* name of the user and host who produced the log;
- *severity:* severity of the log message;
- *message:* the text of the log message.

Logs can be retrieved using the getLogs call. Applications can also use this mechanism to store application-specific performance metrics to allow high-level performance adaptation.

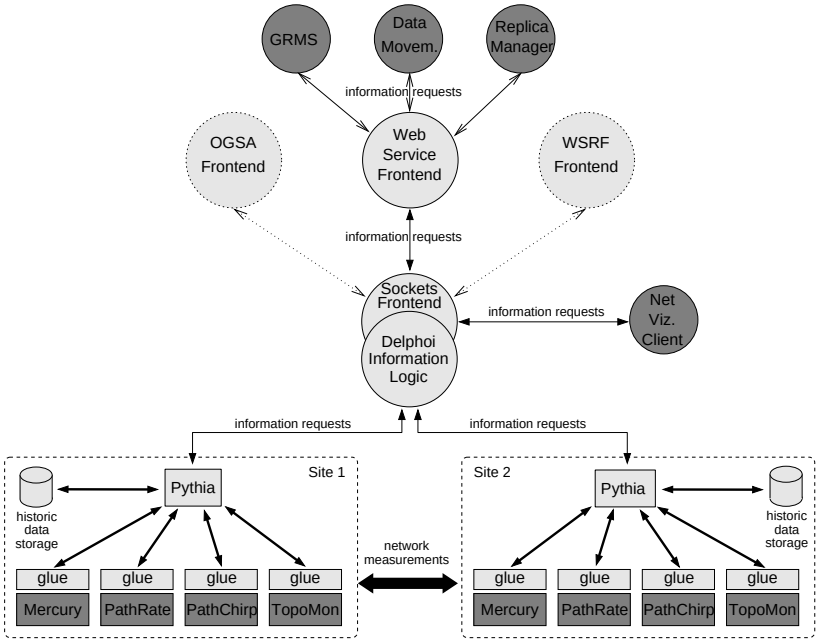


Figure 2. The Delphoi system architecture.

```
String[] getLogs(
    String service,
    String component,
    String origin,
    int severity,
    Calendar startTime,
    Calendar endTime)
```

For each field in the log entries (except message), getLogs accepts a parameter which it will try to match to the entries in logs. Each parameter may contain an exact value, or some regular expression that describes the desired content of the fields. Any matching log entries found that where generated in the specified time interval will be returned.

4. The Delphoi Service Architecture

In this section will describe the architecture of our information framework in more detail. To be portable to various Grid sites, the whole system has been implemented in the Java language, with interfaces to external processes where necessary. For forecasting purposes, we have developed a Java version of the NWS forecaster library [22]. As shown in Figure 2, the framework consists of several components.

The central component in our framework is Delphoi, which serves as a contact point for the clients that are interested in retrieving information from our system. Queries can be sent to Delphoi using its socket or web service frontends. The socket frontend is directly integrated into Delphoi, while the webservice frontend is a separate module, as shown in Figure 2. For the both frontends, we currently have a client that can visualize network performance data.

All queries received by the webservice frontend are simply forwarded to Delphoi using its sockets frontend. Both frontends provide the API which is described in detail in Section 3. The separation of frontends from the core functionality also allows us to easily add frontends to other service architectures like OGSA [10] of WRSF [9].

The third component in our framework is Pythia. Pythia is responsible for gathering the information and answering queries.† Pythia is an internal component of our framework and does not offer any API for external use. Only through the Delphoi can the gathered information be accessed.

A Pythia is continuously collecting information, or metrics, about its environment, and stores these metrics for future reference. Since the amount of metrics collected by a Pythia is large, it is important that a Pythia is "co-located" with the resources where the metrics are produced. In a typical setting, multiple Pythias will be used. In the Gridlab Testbed, for example, one Pythia is running on each site, either on the frontend node of a cluster, or a single CPU of a shared-memory machine.

A Pythia uses separate modules for gathering different types of information. This modular design simplifies the development of new functionality and allows a Pythia to selectively load the modules that are suitable for the site it is installed on. Each module typically consist of some "glue code" to an external tool which performs the measurements.

To collect local information, such as CPU load, memory usage, or the status of the local queing system, Pythia contains a module to communicate with Mercury. When this module is started, it notifies the local instance of Mercury that it wishes to subscribe itself to a certain set of metrics. Mercury will then start collecting these metrics and it will regularly forward them to the Pythia module. When Mercury is running on the frontend of a cluster machine, it is capable of retrieving metrics from all compute nodes. Therefore, a single Pythia installation suffices on a cluster.

Mercury is only capable of collecting metrics produced on a single resource. For the Pythias to collect information about the network connections to other sites, four additional modules are used: *Delay*, *PathRate*, *PathChirp*, and *TopoMon*. Since most network measurements require two sites to cooperate, the Pythias are capable of communicating amongst each other to perform the necessary measurement set-up.

The scheduling of network measurements is done locally. Each module tries to obtain a certain measurement frequency. When a module notices it is time to do a network measurement, it requests a list of all currently-active Pythias from Delphoi. It then contacts each of these Pythias to check if it is running the same module, and whether this module is ready to perform

†In Greek mythology, Pythia was the goddess through the body of which the Delphoi oracle made its predictions.

a measurement. If so, any necessary measurement set-up is done, and the measurement is performed.

Because the measurement scheduling is done locally, *collisions* will occur frequently. A collision occurs when a module contacts another Pythia which is not ready to participate, because it is already busy performing other measurements. (To avoid direct interferences, network modules may only perform a single measurement at a time.) How a collision is handled depends on the measurement frequency of the module. If this frequency is high (e.g., for delay measurements, which are done every few minutes) the module will simply ignore any sites that are not ready to cooperate. They will have to wait until the next run. If the frequency is low (e.g., for capacity measurements, which are typically done only once a day), the module will remember any sites that where busy, and retry them later. It will keep retrying until it manages to perform all measurements, or until the frequency interval has expired and all sites must be re-measured.

The Delay module regularly performs a delay measurement to the other sites. It simply sends one-byte packets back and forth repeatedly and calculates the average time required for one round-trip.

The PathChirp module determines the available network bandwidth between two sites using PathChirp [17]. PathChirp is an active probing tool for estimating the available bandwidth on a communication network path. It is based on the concept of *self-induced congestion*. PathChirp sends series of multiple packets in special patterns, called *chirps*. By rapidly increasing the probing rate within each chirp, PathChirp obtains information from which it can dynamically estimate the available bandwidth.

The PathRate module determines the network bandwidth capacity between two sites using PathRate [7]. PathRate uses packet dispersion methods, in conjunction with statistical techniques, to estimate the capacity of the bottleneck link in the network path.

The final measurement module available to Pythia is TopoMon. This module uses the TopoMon [6] tool to determine the network topology between two sites by means of `traceroute`. Unlike the previous network-related modules, the topology module does not need cooperation from a remote Pythia.

Besides measurement modules, a Pythia may also use several high-level modules. These modules do not perform any measurements, but instead contain the intelligence to convert high level queries into the processing of low level metrics. The TCP options module, for example, can predict the optimal TCP settings required to to transfer a file between two sites, using the file size, network delay, network capacity, and the maximum TCP buffer size on both sites. The TCP options optimization will be explained in more detail in the next section.

5. Data Transfer Time Optimization

In this section we will illustrate how our framework can be used to perform data tranfer optimizations. Data transfers can be optimized by using multiple TCP streams, and modifing the send and receive buffer sizes used by these streams. To determine the optimal settings, good estimations of the network delay and capacity between the sender and the receiver are required. Manually optimizing these settings is hard, and either involves doing separate performance

measurements or trying to find good settings by trial and error. Even assuming the manual optimization has been performed successfully, it may not remain optimal, since both network delay and capacity may change over time. By using the information gathered by Delphoi and the Pythias, this optimization can be automated.

To illustate the use of our framework, we have written a benchmark which transfers a large amount of data between two sites in the Gridlab testbed. The exact amount of data tranfered 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.

To optimize this data transfer, the *estimateTcpOptions* method of Delphoi is used to retrieve a prediction for the optimal TCP settings. As described in Section 3.4, this method requires the source and destination machines, data size, and start time as parameters. Once the *estimateTcpOptions* call arrives at Delphoi, it forwards it to the Pythia on the sending machine.

The Pythia then retrieves the network delay and capacity metrics from its historic data storage, and uses this information to predict the network performance at the given time. It also retrieves information on the maximum TCP send and receive buffer sizes that may be used on both the sending and receiving machine. It then computes the TCP stream buffer size as:

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

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

However, if the TCP buffer size cannot be set large enough, 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.

The Pythia will calculate how many TCP streams are required to fully use the capacity of the network as follows:

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

Both the desired number of streams and TCP buffer size are then returned to Delphoi, which returns them to the benchmark. Finally, the benchmark retrieves a prediction of the data transfer time from Delphoi, which is handled in a similar manner.

To evaluate the predictions, 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 50% less TCP streams. The amount of data transferred in unchanged. 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 three combinations of sites in the Gridlab testbed to run the benchmark at regular intervals. The sites are shown in Table III. The table also shows approximations* for the amount of data transferred, and the network delay, capacity and available bandwidth. The results of the experiments are shown in Figures 3, 4 and 5.

Table III. Selected Sited for Transfer Time Optimization.

Source	Destination	Latency (ms)	Capacity (MBit/s)	Available (MBit/s)	Size (MB)	Remark
fs0.das2.cs.vu.nl	eltoro.pcz.pl	75	97	83	300	Normal link
cluster3.zib.de	helix.bcvc.lsu.edu	155	60	23	90	High delay
fs0.das2.cs.vu.nl	skirit.ics.muni.cz	19	800	610	1024	High bandwidth

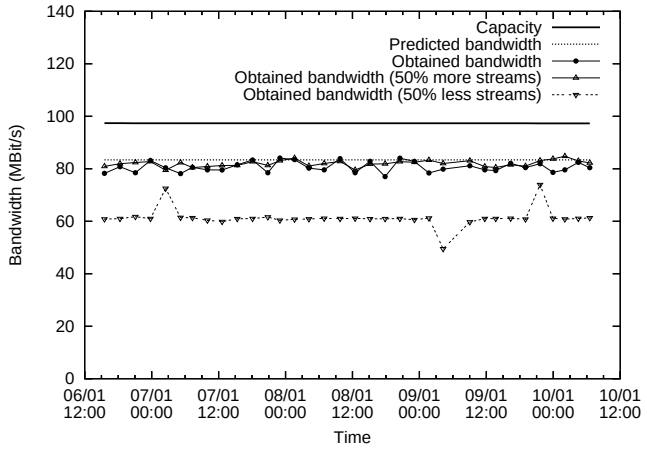


Figure 3. Transfer times from fs0.das2.cs.vu.nl to eltoro.pcz.pl.

Figure 3 shows the results of the transfer optimization test between the fs0.das2.cs.vu.nl (Amsterdam, The Netherlands) and eltoro.pcz.pl (Czestochowa, Poland) 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

*The actual values may vary over time.

bandwidth, while removing 50% of the streams does have a significant impact. Therefore, the TCP settings and transfer time predictions are correct.

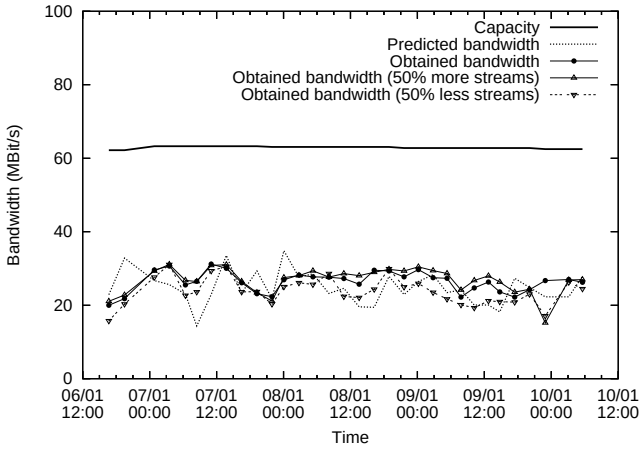


Figure 4. Transfer times from cluster3.zib.de to helix.bcvc.lsu.edu.

Figure 4 shows the results of the transfer optimization test between the cluster3.zib.de (Berlin, Germany) and helix.bcvc.lsu.edu (Louisiana, USA) 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.

The results of our final experiment are shown in Figure 5. This figure shows the results of the optimization test between the fs0.das2.cs.vu.nl and skirit.ics.muni.cz (Brno, Czech Republic). Again, the predicted number of TCP streams is sufficient. Adding 50% more streams only slighty 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.

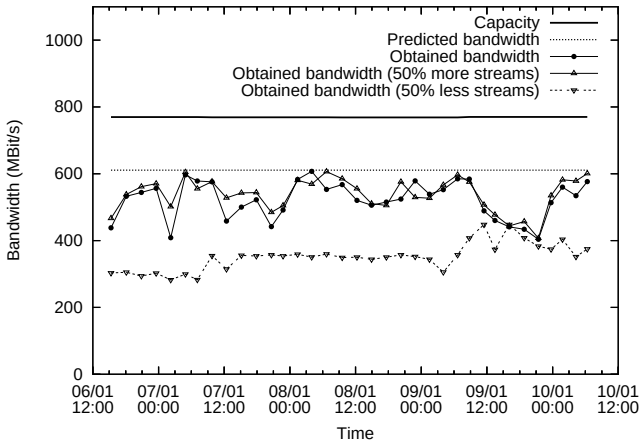


Figure 5. Transfer times from fs0.das2.cs.vu.nl to skirit.ics.muni.cz.

6. Queue Waiting Time Prediction

In this section we will illustrate how our queue information recorded by our framework is used to estimate job waiting times in batch queueing systems. Instead of directly using Delphoi to do experiments, we use the core of the queue information module used by the Pythia's in a simulation.

We 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 behaviour of our system without having to wait for long 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 a Pythia 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 used to estimate the waiting times of future jobs.

Figure 6 shows the result for a PBS log of the fs0.das2.cs.vu.nl[†], covering the period of 1 to 26 March 2004. Because this machine is mainly used for research into parallel and distributed

[†]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 equals size.

computing, most jobs have short lifetime, but 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).

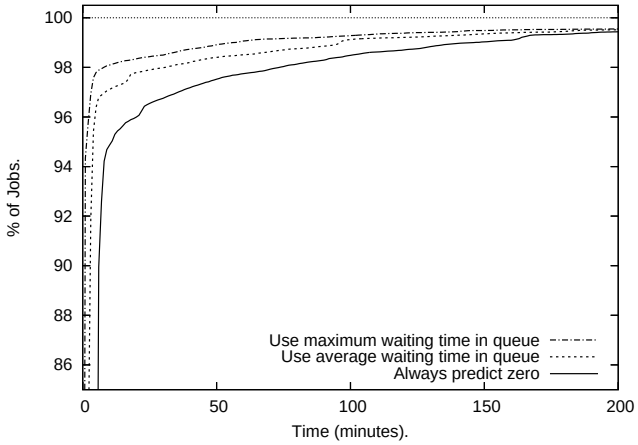


Figure 6. Error in Job waiting time estimation on fs0 (log of march 2004).

Figure 6 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 0. 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 6 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%percent of the jobs is run within 5 minutes of the estimated time.

Figure 7 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

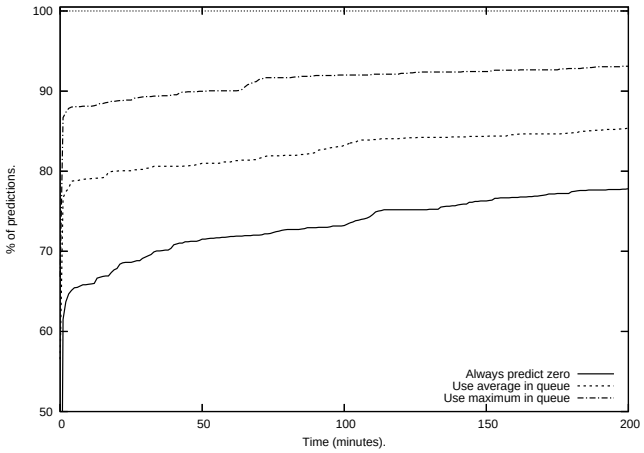


Figure 7. Error in Job waiting time estimation on skirit (from 13 March'04 to 13 April'04).

the maximum 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 6 and 7 indicate that the queue information gathered by Delphoi can be used to provide good predictions for job waiting time. 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.

7. Related Work

Most Grid monitoring systems, like Mercury [11] or NWS [22], have been built according to the Grid Monitoring Architecture (GMA) [21]. Whereas monitoring systems provide dynamically variable performance data, information systems like the Globus MDS [15] or iGrid [4] provide static configuration data. None of them provides a unified interface to all relevant data for given adaptive middleware components.

In contrast, Delphoi implements such a unified interface, combined with translation of low-level measured data to high-level information, and a prediction capability into the near-term future. In fact, Delphoi's predictor is a version of the NWS forecaster library, that we have ported to Java, and made available to all numerical metrics being dealt with by Delphoi.

The Autopilot system [18] was the first to propose a closed-loop architecture for tuning running applications according to actual resource performance information. However, Autopilot's information closely intertwines the application with the remote sensing and controling infrastructure. Using the Delphoi service draws a clear boundary between the

application and the adaptive middleware, allowing to use any Grid service architecture (like OGSA or WSRF) for which a Delphoi frontend has been built.

André et al. propose a generic framework for adaptive software components for Grids [5]. In combination with Delphoi, this framework could retrieve the necessary resource information.

In [13], a collection of articles has been compiled, addressing various aspects of adaptive middleware. Most of these approaches are based upon *reflection* mechanisms that aim at switching implementation strategies. Delphoi is mostly concerned with parametric adaptation of a given strategy. However, in combination with dynamic adaptor loading within the GAT, also switching of strategies or protocols can be implemented using Delphoi.

8. Conclusion

Grid applications need to be aware of the performance variability of the resources they are using. Therefore, Grid middleware is needed both to provide information about the status of the resources and to guide decisions about adapting application performance. Usually, adaptive Grid middleware components re-implement this difficult process over and over again. In this paper, we have presented the Delphoi service that provides unified access to various information sources, translates it to high-level information in combination with prediction into the near-term future. Delphoi has a flexible and extensible system architecture that also supports application-specific metrics and event logging.

Delphoi's architecture separates the core functionality from the service interface, allowing to use different service architectures (like OGSA or WSRF), even simultaneously. On each Grid site, a Pythia process regularly collects performance information and stores it locally. Only upon request, this information is sent to Delphoi. This design, along with the choice of sensors for CPU and network information, results in minimally intrusive operation of our service.

Delphoi is currently being deployed in GridLab's testbed for a variety of performance optimizations used by the Grid Application Toolkit (GAT). In this paper, we presented two example use cases, the fully automated optimization of large data transfers, and the estimation of job waiting times in batch queueing systems. The Delphoi software suite is open source and can be retrieved from www.gridlab.org.

Acknowledgments

This work has been funded by the European Commission, grant IST-2001-32133, as part of the GridLab project. The authors would also like to thank their numerous colleagues from GridLab who have contributed to the development of the system. Andrei Hutanu performed the GridFTP measurements.

REFERENCES

1. Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, Andre Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid – A GridLab Overview. *International Journal on High Performance Computing Applications*, 17(4):449–466, 2003.
2. Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, André Merzky, Rob van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 2004. To appear.
3. Gabrielle Allen, Edward Seidel, and John Shalf. Scientific Computing on the Grid. *Byte*, pages 24–32, Spring 2002.
4. G. Aloisio, M. Cafaro, I. Epicoco, D. Lezzi, M. Mirto, and S. Mocavero. The Design and Implementation of the GridLab Information Service. In *Second International Workshop on Grid and Cooperative Computing (GCC 2003)*. Lecture Notes in Computer Science, Springer, 2003.
5. F. André, J. Buisson, and Jean-Louis Pazat. Dynamic Adaptation of Parallel Codes: toward Self-adaptable Components for the Grid. In V. Getov and T. Kielmann, editors, *Component Models and System for Grid Applications*, pages 145–156. Springer, 2004.
6. Mathijs den Burger, Thilo Kielmann, and Henri E. Bal. TopoMon: A Monitoring Tool for Grid Network Topology. In *International Conference on Computational Science (ICCS 2002)*, pages 558–567, 2002. Published as LNCS Vol. 2330.
7. Constantinos Dovrolis, Parameswaram Ramanathan, and David Moore. What do Packet Dispersion Techniques Measure? In *IEEE Infocom*, 2001.
8. Dietmar Erwin. UNICORE – a Grid Computing Environment. *Concurrency and Computation: Practice and Experience*, 14(13–15):1395–1410, 2002.
9. I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling Stateful Resources with Web Services v. 1.1. White paper, 2004. http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf.
10. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.
11. P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás. P-GRADE: A Grid Programming Environment. *Journal of Grid Computing*, 2:171–197, 2003.
12. Bruce Lowekamp, Brian Tierney, Les Cottrell, Richard Hughes-Jones, Thilo Kielmann, and Martin Swany. A Hierarchy of Network Performance Characteristics for Grid Applications and Services. GFD-R-P.023 (Proposed Recommendation), Global Grid Forum, 2004.
13. Gul Agha (Editor). Special Issue on Adaptive Middleware. *Communications of the ACM*, 45(6), 2002.
14. W. Allcock (Editor). GridFTP: Protocol Extensions to FTP for the Grid. GFD-R-P.020 (Proposed Recommendation), Global Grid Forum, 2003.
15. The Globus Alliance. http://www.globus.org/.
16. The GridLab Project. GridLab Resource Management System (GRMS), 2003. http://www.gridlab.org/grms/.
17. Vinay Ribeiro, Rudolf Riedi, Richard Baraniuk, Jiri Navratil, and Les Cottrell. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop (PAM 2003)*. NLANR, 2003.
18. R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proc. High-Performance Distributed Computing (HPDC)*, Chicago, IL, July 1998.
19. H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Supercomputing (SC2000)*, page 38, 2000.
20. The GridLab Project. GridLab Data Management Services, 2003. http://www.gridlab.org/data/.
21. Brian Tierney, Ruth Aydt, Dan Gunter, W. Smith, Martin Swany, Valerie Taylor, and Rich Wolski. A Grid Monitoring Architecture. GFD-I.7 (Informational), Global Grid Forum, 2002.
22. R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems*, 15(5–6):757–768, 1999.