Middleware Adaptation with the Delphoi Service

Jason Maassen, Rob V. van Nieuwpoort, Thilo Kielmann, Kees Verstoep
Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{jason,rob,kielmann,versto}@cs.vu.nl
http://www.gridlab.org/delphoi/

***Abstract*— Grid middleware needs to adapt to changing resources for a large variety of operations. Currently, information about Grid resources can be retrieved from various monitoring and information systems. However, the available information is rather low level and also dispersed across many individual sources. In this paper, we present the Delphoi service that provides a unified interface to all necessary information, on an abstraction level that matches the adaptation needs of middleware services. 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.**

I. 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 that are 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 access to information, Delphoi provides translation to high-level information in combination with prediction[1] 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 optimizations. In this paper, we present one example use case, the fully automated optimization of large file transfers with GridFTP.

In Section II, we outline the need for behavior adaptation by GridLab's Grid Application Toolkit (GAT). Section III presents the Delphoi service emphasizing its high-level interface to resource-related data. Section IV presents Delphoi's architecture and distributed implementation. In Section V we present one example use case: the fully automated adaptation of GridFTP. We discuss related work in Section VI before we conclude in Section VII.

[1]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.

II. 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 [3] or Unicore [4]. For more advanced functionality, GridLab also provides higher-level services like the GRMS resource broker [5] or the services for remote file access, data movement, and replication management [6]. 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, the following adaptation use cases are the most important ones that are dealt with by GridLab:

- Data transfer protocol optimization.
  For job execution on a Grid node, data and program files need to be transferred to 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 [7]) and on the network performance characteristics [8].

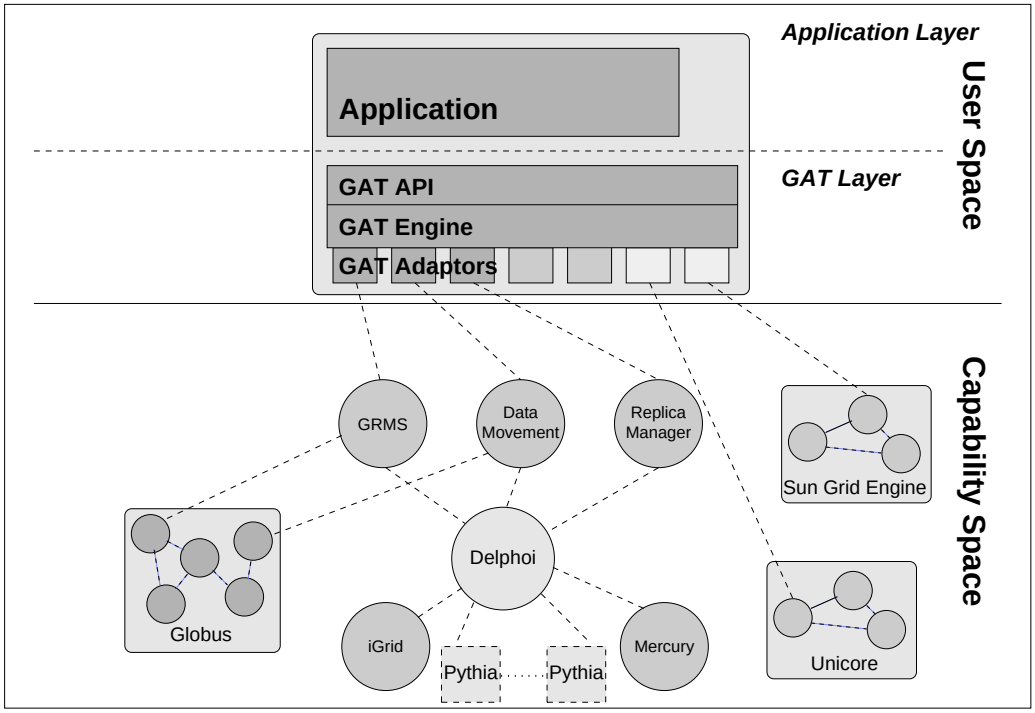

Fig. 1. GridLab's GAT framework software architecture.

- 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 [9] 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.

III. 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 IV, we currently provide RMI and Web Service implementations of this interface.

A. 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:

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

The *MetricInfo* object contains information about a particular metric, such as its name and its parameters. A metric

name could for instance be “freeDiskSpace”, while a parameter could then be the physical device that is measured (“/dev/hda”).

B. Low-level Resource and Network Information

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 returns predictions based on historic data. A time frame is passed to many of the calls presented in this section, using start and end times. The time frame can be in the past, the future, or it could start in the past and end in the future. If the start and end times are *null*, the most recent data available will be returned. A metric for a single resource can be retrieved with the following call:

```
String estimateMetric(
    String hostName,
    MetricInfo metric,
    String operation,
    Calendar startTime,
    Calendar endTime)
```

The *MetricInfo* object has been described above. It can specify “host.load”, for instance, to obtain information about the CPU load on the host. This particular metric does not require additional parameters. The *operation* parameter can be “min”, “mean” or “max”, indicating the minimum, mean or maximum host load during the given time frame. The result of this method is a string, because the actual data type that is returned depends on the requested metric.

The estimateMetric call can also be used to retrieve low-level network performance data, such as bandwidth, capacity or latency. The “hostName” parameter then represents the source, while the destination is passed as a parameter to the *MetricInfo* object. A list of network metrics that are currently supported (in accordance with [8]), is shown in Table I. Besides network metrics, Delphoi also supports single-resource metrics such as CPU load, memory and disk usage, and queuing information. These metrics are provided by Mercury [10] and iGrid [11], the monitoring and information systems developed in GridLab.

TABLE I
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
hoplist	hoplist of network between two sites annotated with hop.delay.oneWay values

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

```
public String[] estimateMetricForMultipleHosts(
    String[] hostNames,
    MetricInfo metric,
    String operation,
    Calendar startTime,
    Calendar endTime)
```

This method is similar to the previous one, but it can be used to simultaneously retrieve a single 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.

Finally, Delphoi allows retrieving raw, unprocessed, metric data:

```
String[] getRawMeasurementData(
    String hostName,
    MetricInfo metric,
    Calendar startTime,
    Calendar endTime)
```

This data can be used for debugging or visualization purposes.

C. High-level Network Information

In addition to low-level metrics, Delphoi provides additional value-added 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 kilobytes that has to be transferred, it predicts the TCP send buffer size and the number of parallel TCP streams (as proposed by [12]) that should be used for maximal performance. The “transferMethod” input parameter indicates the transfer method that will be used (e.g. GridFTP [7]). 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 V, we will show a scenario that uses this call to optimize GridFTP file 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)
```

Like the previous one, this call estimates the time it will take to transfer an amount of data from one grid site to another. For efficiency reasons, however, the call can predict the time for multiple destinations. This call 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.

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

This call mirrors the previous one: it predicts the data transfer time to a given destination for multiple source sites. Within GridLab, this call 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.

D. 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:

```
QueueConf getQueueConf(Queue queue)
```

GetQueueConf 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.

E. Logging

Delphoi also provides an interface to retrieve logging information. Services or applications write log entries locally using Mercury[10]. 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.

```
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.

IV. 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 [13]. As shown in Figure 2, the framework consists of several components.

The central component in our framework is Delphoi, containing the intelligence to convert high level queries into the processing of low level metrics. For example, to predict the time it takes to transfer a file between two sites, Delphoi knows that both the file size and network bandwidth between the two sites are required.

Queries can be sent to Delphoi using its RMI or web service frontends. The RMI frontend is directly integrated into Delphoi, while the webservice frontend is a separate module, as shown in Figure 2. For the RMI frontend, we currently have a client that can visualize network performance data.

All queries received by the webservice frontend are simply translated to RMI queries and forwarded to Delphoi. Both frontends provide the API which is described in detail in Section III. The separation of frontends from the core functionality also allows us to easily add frontends to other service architectures like OGSA [14] of WRSF [15].

The third component in our framework is Pythia. Pythia is responsible for gathering the information that Delphoi needs to answer queries.[2] Pythia is an internal component of our framework and does not offer any API for external use. Only Delphoi is able to access the gathered information.

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 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

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

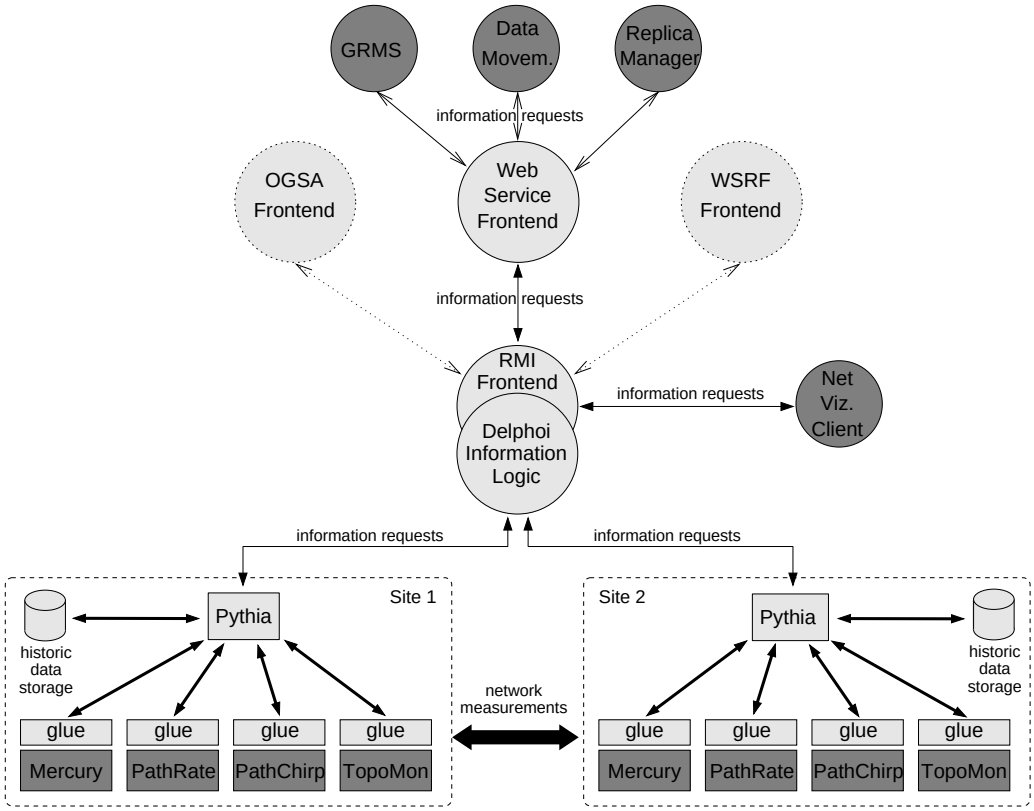

Fig. 2. The Delphoi system architecture.

bandwidth between two sites using PathChirp [16]. 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 [17]. PathRate uses packet dispersion methods, in conjunction with statistical techniques, to estimate the capacity of the bottleneck link in the network path.

The final module available to Pythia is TopoMon. This module uses the TopoMon [18] 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.

V. Example: Transfer Time Optimization with GridFTP

In this section we will describe how our framework can be used to optimize data transfer times. Data has been transferred from a single site in the Gridlab testbed (litchi.zib.de) to seven other sites, using GridFTP as the transfer mechanism. As shown in Table II, the amount of data transferred varies per destination between 50 and 310 megabytes.

To optimize the data transfer, GridFTP allows the user to use multiple TCP streams and modify the send and receive buffer sizes of each stream. To determine the optimal settings, a good estimation 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.

In our experiment, the data is initially transferred using an untuned GridFTP. Figure 3 shows the required transfer time to each of the sites. Next, to optimize the transfer, the *estimateTcpOptions* method of Delphoi is used. As described in Section III-C, this method requires the source and destination machines, data size, and start time as parameters.

Once the *estimateTcpOptions* call arrives at Delphoi, it requests an estimate of the network delay from the Pythia on the sending machine and an estimate of the network capacity from the Pythia on the receiving machine. The Pythias then retrieve the appropriate metrics from their historic data storage, and use this information to predict the network performance at the given time. This information is returned to Delphoi. Delphoi also retrieves information on the maximum TCP send and receive buffer sizes that may be used on both machines.

TABLE II

TRANSFER TIME OPTIMIZATION WITH GRIDFTP (SOURCE HOST LITCHI.ZIB.DE).

Destination Host	Transferred Data Size (MBytes/sec.)	Non-Optimized (seconds)	Optimized (seconds)	parallel Streams	TCP Buffer Size (KBytes)
skirit.ics.muni.cz	310	126	30	9	64
hitcross.lrz-muenchen.de	310	134	150	5	128
gridentry.uni-paderborn.de	50	77	36	6	64
rage1.man.poznan.pl	310	319	61	9	128
n0.hpcc.sztaki.hu	310	164	36	9	64
fs0.das2.cs.vu.nl	300	163	39	5	128
mike4.lsu.edu	102	284	78	16	64

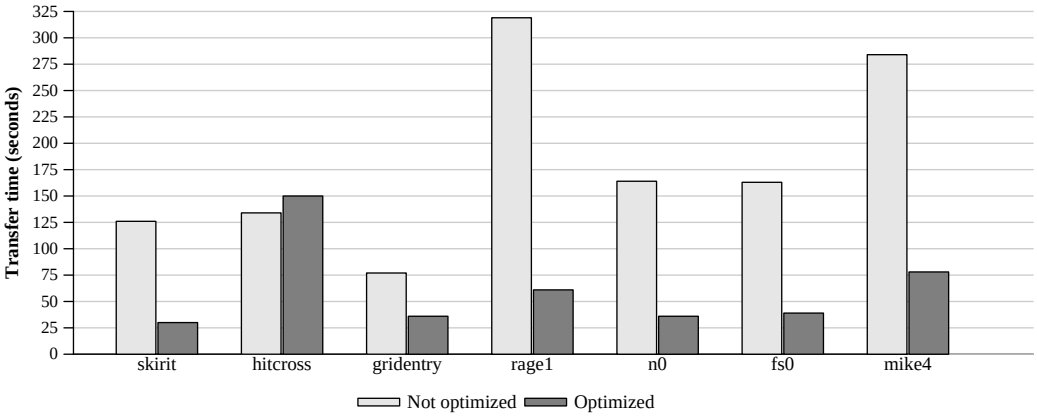

Fig. 3. Transfer times from litchi.zib.de.

It 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, Delphoi 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. Delphoi 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.

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

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

Both the desired number of streams and TCP buffer size are then returned to the client.

By using these TCP option estimations, we were able to significantly improve the performance of the GridFTP data transfer. The result of this optimized data transfer is shown in Figure 3. In all cases but one, the transfer times could be reduced significantly. Only with hitcross, the concurrency induced by parallel streams actually increased transfer times slightly, because this machine has only remote NFS file systems that seemingly dominate transfer performance to this machine.

VI. RELATED WORK

Most Grid monitoring systems, like Mercury [10] or NWS [13], have been built according to the Grid Monitoring Architecture (GMA) [19]. Whereas monitoring systems provide dynamically variable performance data, information systems like the Globus MDS [3] or iGrid [11] 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 [20] 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 [21]. In combination with Delphoi, this framework could retrieve the necessary resource information.

In [22], 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.

VII. 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 one example use case, the fully automated optimization of large file transfers with GridFTP. We have shown that Delphoi can automatically tune such file transfers, minimizing transfer times. Other use cases show similar results, applied to different adaptation goals. 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] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, "Enabling Applications on the Grid – A GridLab Overview," *International Journal on High Performance Computing Applications*, vol. 17, no. 4, pp. 449–466, 2003.

[2] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer, "The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid," *Proceedings of the IEEE*, 2004, to appear.

[3] The Globus Alliance, http://www.globus.org/.

[4] D. Erwin, "UNICORE – a Grid Computing Environment," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13–15, pp. 1395–1410, 2002.

[5] The GridLab Project, "GridLab Resource Management System (GRMS)," 2003, http://www.gridlab.org/grms/.

[6] The GridLab Project, "GridLab Data Management Services," 2003, http://www.gridlab.org/data/.

[7] W. Allcock (Editor), "GridFTP: Protocol Extensions to FTP for the Grid," GFD-R-P.020 (Proposed Recommendation), Global Grid Forum, 2003.

[8] B. Lowekamp, B. Tierney, L. Cottrell, R. Hughes-Jones, T. Kielmann, and M. Swany, "A Hierarchy of Network Performance Characteristics for Grid Applications and Services," GFD-R-P.023 (Proposed Recommendation), Global Grid Forum, 2004.

[9] G. Allen, E. Seidel, and J. Shalf, "Scientific Computing on the Grid," *Byte*, pp. 24–32, Spring 2002.

[10] 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*, vol. 2, pp. 171–197, 2003.

[11] 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.

[12] 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)*, 2000, p. 38.

[13] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing," *Future Generation Computing Systems*, vol. 15, no. 5–6, pp. 757–768, 1999.

[14] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, vol. 15, no. 3, 2001.

[15] 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.

[16] V. Ribeiro, R. Riedi, R. Baraniuk, J. Navratil, and L. Cottrell, "pathChirp: Efficient Available Bandwidth Estimation for Network Paths," in *Passive and Active Measurement Workshop (PAM 2003)*. NLANR, 2003.

[17] C. Dovrolis, P. Ramanathan, and D. Moore, "What do Packet Dispersion Techniques Measure?" in *IEEE Infocom*, 2001.

[18] M. den Burger, T. Kielmann, and H. E. Bal, "TopoMon: A Monitoring Tool for Grid Network Topology," in *International Conference on Computational Science (ICCS 2002)*, 2002, pp. 558–567, published as LNCS Vol. 2330.

[19] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski, "A Grid Monitoring Architecture," GFD-I.7 (Informational), Global Grid Forum, 2002.

[20] 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.

[21] F. André, J. Buisson, and J.-L. Pazat, "Dynamic Adaptation of Parallel Codes: toward Self-adaptable Components for the Grid," in *Component Models and System for Grid Applications*, V. Getov and T. Kielmann, Eds. Springer, 2004, pp. 145–156.

[22] G. Agha (Editor), "Special Issue on Adaptive Middleware," *Communications of the ACM*, vol. 45, no. 6, 2002.