IST-2001-32133
GridLab - A Grid Application Toolkit and Testbed

Design of Adaptive Components

Author(s):	Thilo Kielmann, Rob van Nieuwpoort, Jason Maassen
Document Filename:	GridLab-7-DAC-0002-DesignReport
Work package:	WP7: Adaptive Grid Components
Partner(s):	Vrije Universiteit (VU)
Lead Partner:	Vrije Universiteit (VU)
Config ID:	GridLab-7-DAC-0002-1.0-DRAFT-A
Document classification:	IST

Abstract: This report describes the design of the adaptive components used in the GridLab project. These adaptive components are designed to handle the use cases described in the deliverable document GridLab-7-UCR-0001-UseCasesReport.

January 6, 2003

Contents

1 Introduction **2**

2 Design **2**
2.1 Interface to the application 2
2.2 Interface to the user/portal 3
2.3 Interface to the monitoring system 4
2.4 Main parts of an adaptive component 4

3 Access to the Monitoring System **5**
3.1 Interface of the Adaptive Service 5
3.1.1 getServiceDescription 6
3.1.2 rank_resources 6
3.1.3 estimate_transfer_time 7
3.1.4 log_data_transfer 8
3.1.5 estimate_usage 8
3.1.6 estimate_multiple_usage 9
3.1.7 estimate_network_graph 10

4 Integration with Use Cases **11**
4.1 Throughput Optimization of Triana Flow Graphs (1) 11
4.2 Throughput Optimization of Triana Flow Graphs (2) 11
4.3 Parameter Adaptation for Parallel I/O 11
4.4 N-to-M Data Transfer Time Estimation 12
4.5 Remote Data Visualization 12
4.6 Communication Adaptation of Distributed Simulations (1) 12
4.7 Communication Adaptation of Distributed Simulations (2) 13
4.8 Distributing Computation across Heterogeneous Processors 13
4.9 Compute Time Estimation 13

1 Introduction

Grid resources are highly dynamic: network connections have varying bandwidth and latency, available CPU resources come and go. Static (inflexible) strategies and implementations are thus not suitable in Grid environments. Adapting application behavior (together with middleware layers) thus is inevitable.
In WP7, we are generalizing the basic techniques of gathering relevant monitoring data, correlating this data with performance models (specific to GAT modules), and short-term prediction of application performance, finally leading to behavior adaptation. The WP will develop generic components for implementing adaptive behavior.
The goal of this work package is to provide adaptive application components to other modules of the application toolkit. Common to these components is that they take dynamic information about Grid resources (network status, CPU availability, memory footprint, etc.) as input to module-specific performance-prediction models and implement adaptive strategies that let applications efficiently use the given resources. For grid-aware applications, this general mechanism is used for numerous purposes like data access, task scheduling, and dynamic application reconfiguration. Although use-case specific APIs and event models are necessary, the basic techniques are similar. The objective is to develop adaptation components that will be instantiated to their use from other modules of the GAT. For the input data, this WP will rely on the work performed in WP11 (monitoring).
In this report, we describe the basic design of the anticipated, adaptive components. We identify core functionality and describe the interface to the monitoring system. Finally, we outline how this design shall be used for the individual use cases that have been identified in the report GridLab-7-UCR-0001-UseCasesReport.

2 Design

Figure 1 shows the interactions of an adaptive component with other components involved in an application run. Before going into further detail, the following clarifications seem appropriate:

1. Throughout the project, the term *adaptive component* has been coined. We continue using it to avoid confusion. However, the term *adaptation component* might be a better description, actually: software components that implement adaptive behavior in behalf of the application.

2. Whenever we use the term *application*, actually we refer to a resource-critical part (module, component, thorn, unit, etc.) of the application (e.g., a GAT module) that is made adaptive by adding one of the adaptive components.

2.1 Interface to the application

Core of the operation is the control loop being established between the adaptive component and the application being controled by it. While the application is running, the adaptive component receives data about the application behavior. In other contexts, this is also refered to as *application instrumentation*. Here, we restrict instrumentation to the critical application component. In reaction to the instrumentation data, the adaptive component controls the application.
The API and event model for this control loop is application-specific. The following event models are possible:

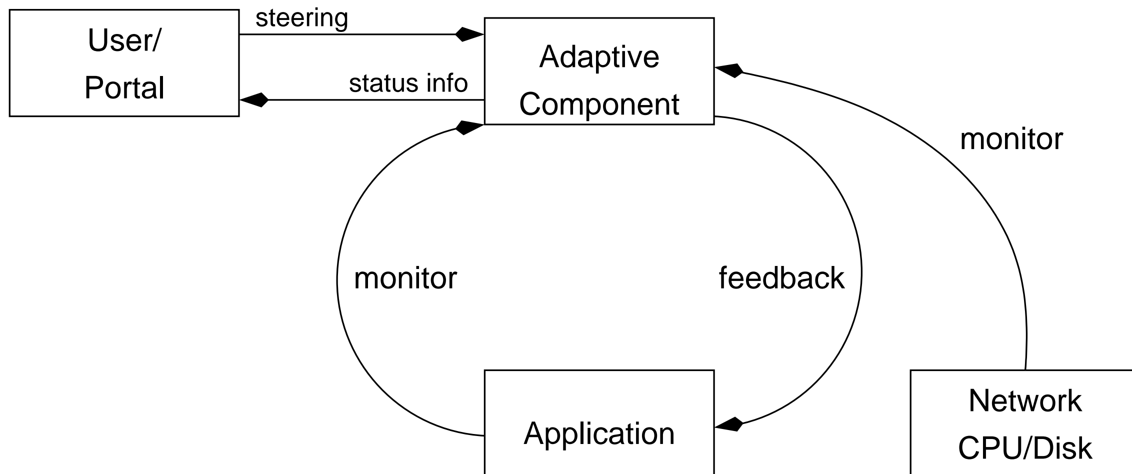

Figure 1: Interactions of an Adaptive Component

Passive Component. The adaptive component is completely passive; it is implemented as a library function. The application invokes the adaptive component whenever necessary. Here, the instrumentation data is passed to the adaptive component as function parameters. The adaptive component returns a desired value, causing the application to adapt its behavior accordingly.

Active Component. The adaptive component has its own activity (thread, process, unit, etc.). The adaptive component monitors both the application behavior and the critical resources (e.g., networks). Whenever the adaptive component finds adaptation to become necessary, it invokes the controled application component to trigger some change of behavior.

This model allows two variations for application instrumentation:

1. Passive instrumentation; the application invokes the adaptive component whenever instrumentation data is available.
2. Active instrumentation; the adaptive component invokes the application to retrieve instrumentation data.

Hybrid Component. The adaptive component has its own activity, but uses it only to collect monitoring data proactively. The application still invokes the adaptive component as a library function to retrieve adaptation information whenever desired (without being interrupted).

In other words, a hybrid, adaptive component behaves as a library function that implements latency hiding for retrieving monitoring data.

2.2 Interface to the user/portal

This interface is optional and also application dependent. In a minimalistic scenario, the adaptation goal merely is *"maximize speed"*. In this case, no interface to the user or portal is necessary. However, in the following two cases, such an interface may become useful:

1. Reporting application progress to the user. The adaptive component might have the best picture of the application's progress, so it might report this to the user or portal. Another

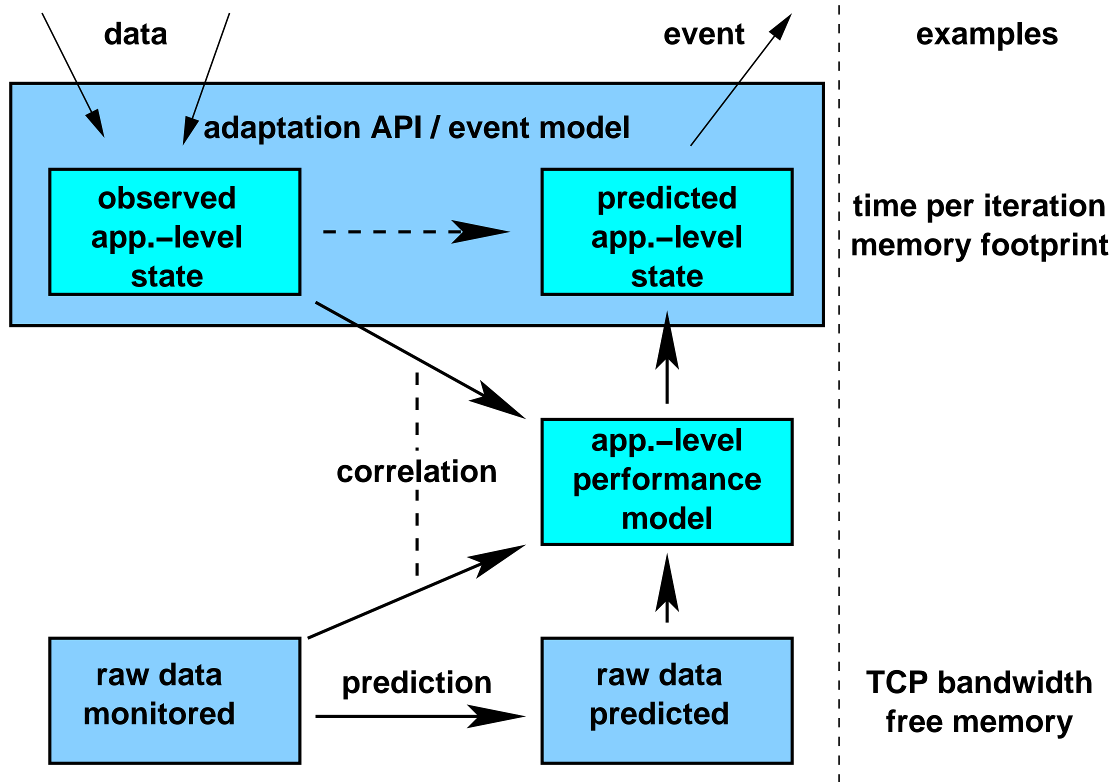

Figure 2: Adaptation mechanisms

entity to communicate with for such purposes might be an *application status manager* component.

2. In some cases, the adaptation goal might not be obvious. Instead, tradeoffs might be controled by the user. For example, in a remote visualization scenario, the user might favor image quality over a frame rate, or visualization delay. These user decisions are needed to steer the adaptive component.

2.3 Interface to the monitoring system

This interface will be described in the following section.

2.4 Main parts of an adaptive component

Figure 2 shows the data being processed inside an adaptive component and its transformations being performed. The major goal of any adaptive component is to observe the application status, predict its future status based on the given resources, and trigger appropriate actions by sending events to the application.
The event model for retrieving instrumentation data and for sending events already has been desribed above. The *direct* prediction of the future application status might not be possible, however, as this depends on many factors, among which are both underlying resources and the application itself.
Instead, each contributing factor can be predicted separately. The work on the *Network Weather Service* (NWS) [1] has shown the feasibility of this approach. Our adaptive components will thus

use elementary prediction mechanisms (like regression models) both for the underlying resources (like CPU or network) and where necessary also for the application behavior itself.
The central mechanism for predicting the application behavior is based on the following: Given the current resource state R, and the current application behavior A; the predicted resource state (for a given near-future point in time t) is $R_p(t)$, compute $A_p(t)$ by correlating R and A.
This application behavior prediction relies on an application-level performance model that can express (approximatively) $A(t)$ depending on $R(t)$. This performance model belongs to the application-specific part of an adaptive component.

3 Access to the Monitoring System

The adaptive components developed by WP7 (described in the use cases of the report GridLab-7-UCR-0001-UseCasesReport) all provide application specific adaptation mechanisms. However, the implemention of these mechanisms often requires information that is not directly available to the application (such as global or historic information). Therefore, we have designed an *adaptive service* which is capable of providing such information to the adaptive components. (As with the general design, this name has been chosen in accordance with early-stated terminology within the project. A name like *adaptation information service* might be more precise, but also more complicated and confusing.) The design of this service is shown in Figure 3.
The adaptive service consists of two components, the Adaptive Components Service (ACS) and the Local Adaptive Components (LAC). The ACS provides an interface to query the adaptive service. While some queries may be handled directly by the ACS, others may need detailed historical information or up-to-date measurements that are only available on the resources themselves. The LAC has been designed for this purpose.
The LAC is capable of using the local monitoring system (the LM and MM boxes), to continuously collect data about the resource and applications running on it (load information, queue lengths, network bandwidth and topology to other machines, etc.). This data is processed and may be stored locally for future reference. Since the amount of data transferred between the LAC and the monitoring system can be large, it is important that they are co-located. Therefore, like the monitoring system, the LAC generally runs on the resource itself (the frontend machine of a cluster, for example).
When a query is submitted to the ACS, asking to predict the load on a machine, for example, it can be forwarded to the LAC which has the information required produce an answer.

3.1 Interface of the Adaptive Service

The current ACS interface supports the following queries. This interface is intended mainly to serve a rapid-prototype implementation. It is subject to change during the project lifetime for the following reasons:

1. As the underlying technology (OGSA) is currently evolving, changes might have to be incorporated.
2. As GridLab-wide security mechanisms have been established, these might have to be incorporated.
3. As GridLab is application driven, actual experience with the use cases might require modifications to the service interface as well.

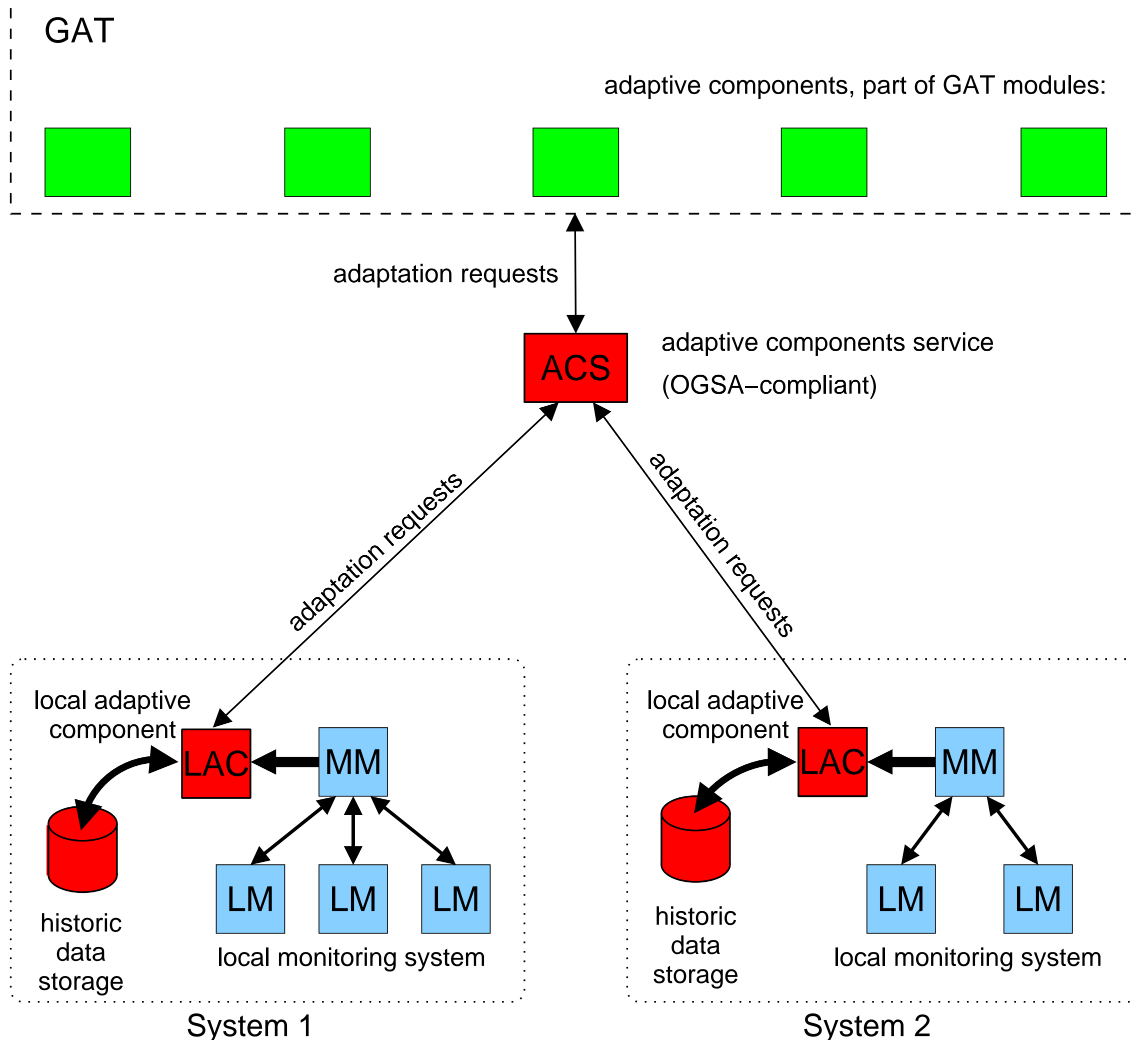

Figure 3: Access to the Monitoring System

3.1.1 getServiceDescription

```
/* A call to identify our service (OGSA requirement) */
int Adaptive__getServiceDescription(xsd__string* result);
/*
```

3.1.2 rank_resources

```
rank_resources:
---------------


decription:
This call will sort a list of resources using the predicted
performance of the application (best performance first).  If
known, each resource in the list will be annotated with the
predicted run time and data write (or checkpoint) time of the
application.  It is assumed that all resources are 100%
```

```
available.

input:
* res: list of resources descriptions
- resource ID
- available flops, network and IO bandwidth, etc.

* app: application description (ask portal folks)
- name/URI of application
- input set/problem size of the application
- If known, (location of) performance model which
  Describes resource utilization of application
  (needed flops, network and IO usage, etc.).


output:
* out: The same list of resource descriptions, but now sorted, and
       annotated with predicted run times (-1 if not known) and the
               predicted time it will take to write the application output
               (or checkpoint) to disk.  i.e., a list of (resource, run
               time, disk write time), but ranked on total time.
*/
int Adaptive__rank_resources(
struct Adaptive__resource_info_vector_t* res,
struct Adaptive__application_info_t* app,
struct Adaptive__resource_info_vector_t** out);
```

3.1.3 estimate_transfer_time

```
/* estimate_transfer_time
estimated transfer time of data between grid sites
--------------------------------------------------

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

input:
* boolean: do we need to substract gridlab traffic ?
* source site
* destination site list
* data size
* start time

output:
* a list of (destination site, required time to transfer data)
*/
```

```
int Adaptive__estimate_transfer_time(
xsd__int substract_gridlab_traffic,
xsd__string source_name,
struct Adaptive__string_list_t* destination_names,
xsd__int data_size,
xsd__dateTime start_time,
struct Adaptive__int_list_t** res);
```

3.1.4 log_data_transfer

```
/*
log gridlab data transfer
-------------------------

description:
This call can be used to inform the adaptive component of
(large) data transfers.  This way, the adaptive component can
filter out gridlab traffic when making predictions.  This call
is thus optional, and is only used to make better predictions
possible.

input:
* source site
* destination site
* data size
* start time
* end time

output:
* none
*/
int Adaptive__log_data_transfer(
xsd__string source_name,
xsd__string destination_name,
xsd__int data_size,
xsd__dateTime start_time,
xsd__dateTime end_time,
xsd__int* dummy);
```

3.1.5 estimate_usage

```
/*
estimate future usage of single resource
----------------------------------------

input:
* boolean: filter out gridlab jobs
* resource
* metric
```

```
* desired operator (MIN, MAX or MEAN)
* start time
* duration

output:
* a single number describing the usage of the resource
*/
int Adaptive__estimate_usage(
xsd__int substract_gridlab_jobs,
struct Adaptive__resource_info_t* res,
xsd__string metric,
xsd__int* operation,
xsd__dateTime start_time,
xsd__duration duration,
xsd__string* out);
```

3.1.6 estimate_multiple_usage

```
/*
estimate future usage of multiple resources
-------------------------------------------

input:
* boolean: filter out gridlab jobs
* list of resources
* list of metrics that have to be predicted
* start time
* duration

output:
for all resources
    for all metrics
a time series: a list of (time, data, accuracy)




possible metrics are for instance:
 * CPU load
 * disk
 * network bandwidth
 * network latency
 * memory load
 * local queue waiting time
*/
int Adaptive__estimate_multiple_usage(
xsd__int substract_gridlab_jobs,
struct Adaptive__resource_info_vector_t* res,
struct Adaptive__string_list_t* metric,
```

```
xsd__dateTime start_time,
xsd__duration* duration,
struct Adaptive__resource_metrics_list_t** out);
```

3.1.7 estimate_network_graph

```
/*
Estimate the network performance between a set of resources.
------------------------------------------------------------

description:
This call is used to estimate the network performance between a set of resources.
For instance:

in: (VU, UvA, Delft, Leiden), "bandwidth", now, 1 hour
out: a graph:

            8;8
        VU------UvA
        |\      /|
        | \6;6 / |
        |  \  /  |
     4;4|   \/   |1;2
        |   /\   |
        |  /  \  |
        | /2;3 \ |
        |/      \|
      Delft----Leiden
           5;5

        The output graph is an annotated graph of the topology between the resources.
The graph is directed: (outbound; return)


input:
* boolean: filter out gridlab jobs
* A set of resources
* The network metric that has to be returned (e.g., bandwidth, latency, capacity).
* The timeframe of the measurement (start_time, duration).

output:
* a directed graph annotated with values for the requested metrics.
*/
int Adaptive__estimate_network_graph(
xsd__int substract_gridlab_jobs,
struct Adaptive__resource_info_vector_t* in,
xsd__string metric,
xsd__dateTime start_time,
xsd__duration* duration,
```

```
struct Adaptive__resource_info_graph_t** out);
```

4 Integration with Use Cases

We use the information provided by the adaptive service to implement the adaptive components. We will now give a short description of the design of the adaptive components required for each of the uses cases.

4.1 Throughput Optimization of Triana Flow Graphs (1)

This scenario requires an adaptive component which is capable of mapping the units of a Triana flow graph to a set of machines. The problem is to assign the units to the machines such that the overall system computes as fast as possible (with maximal throughput). Care needs to be taken of machines having different computing speeds and units having different computational needs. In this scenario, we assume the Triana graph to be computation-bound, so network-related aspects like getting data fast enough from one machine to another can be ignored safely.
The adaptive component (implemented as a Triana unit) receives two input parameters: the set of target machines and a description of the Triana flow graph (containing the units, their computational requirements, and the connections between them). By using the `estimate_multiple_usage` call offered by the adaptive service, the computational performance for each of the machines can be estimated. This information is used to determine an optimal mapping of units to machines. The adaptive component returns a vector containing <unit, machine> entries.

4.2 Throughput Optimization of Triana Flow Graphs (2)

This is an extention of the previous scenario with network-related aspects. Now, we also take the available network bandwidth between Triana units (on different machines) into account. This scenario assumes that large data sets are sent between the individual units.
The adaptive component (implemented as a Triana unit) receives two input parameters: the set of target machines and a description of the Triana flow graph (containing the units, their computational requirements, the connections between them, and the bandwidth requirements for each connection). By using the `estimate_multiple_usage` call offered by the adaptive service, the computational performance for each of the machines can be estimated. By using the `estimate_network_graph` call offered by the adaptive service, the network performance between the machines can be estimated. This information is used to determine an optimal mapping of units to machines. The adaptive component returns a vector containing <unit, machine> entries.

4.3 Parameter Adaptation for Parallel I/O

In this scenario, a parallel application is running on a single, parallel machine, performing I/O to locally attached disks. The choice of filesystem (local per CPU or remote via a LAN network) and the type of I/O layer strongly influence application performance. This has to be selected individually for each parallel machine that has actually been allocated to an application. Additionally, the CPU stride (the selection of CPUs that actually perform I/O) for parallel I/O needs to be taken into account.
An adaptive component for this scenario can estimate I/O behavior directly via instrumenting the local file system operation of a currently running application, without involvement of the adaptive service interface. However, it might be beneficial to employ the adaptive service to retrieve historic data from previous application runs on the same platform.

4.4 N-to-M Data Transfer Time Estimation

In this scenario, a parallel (e.g., Cactus) run migrates from a machine with N CPUs to another machine with M CPUs. The time needed to transfer the application data between the machines needs to be estimated in advance, for multiple purposes. First, a migration target machine may be selected by the time it takes to migrate to it (in behalf of the GRMS). Second, after a migration target has been chosen, one of possibly many network paths (e.g., testbed networks) may be selected. Third, after target machine and network has been selected, the transfer software/protocol may be selected (like GridFTP vs. scp). 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.

In this scenario, the adaptation component is passive and can be called from the process that triggers the migration. This process provides a set of machines to migrate to. Using the `estimate_network_graph` call from the adaptive service, the adaptive component retrieves a bandwidth graph describing the network between the current and the target machines. This graph is then used to estimate the transfer time, which is returned.

It is also possible to providing multiple sets of machines. The adaptive component will then rank them according to estimated transfer time. Selection between multiple communication protocols can be modeled via multiple application-level performance models.

4.5 Remote Data Visualization

In this scenario, a user wants to visualize the output of a running, remote computation (like a Cactus run). The critical resource is the network bandwidth that limits either the quality or the delay of the visualization, or even both. The key to this adaptation problem is to perform visualization using *progresive 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 receives network bandwidth data from the monitoring system and uses this to produce short-term bandwidth predictions. From the application, it receives transfer times of image data over the measured link. It can then correlate the transfer time of image data with the measured link bandwidth, and provide an estimate for the image data volume that can be sent within a given time limit. This estimate can be used by the visualization software to control the generated image resolution.

4.6 Communication Adaptation of Distributed Simulations (1)

In this scenario, a parallel Cactus simulation runs distributed across multiple machines. The simulation data is distributed across the individual nodes, where at the machine boundaries the array borders are mirrored to the respective neighbor machines in so-called ghostzones. Per simulation timestep, one ghostzone is necessary for computing. By increasing the number of ghostzones per node, the number of computable timesteps (before the next, updated ghostzone has to be received) can be increased, at the expense of replicating the computation for the ghostzone data to both respective neighbors. The optimization problem is to find a tradeoff between additional computation and synchronization overhead that minimizes execution time.

The adaptation component can use the `estimate_transfer_time` call in the adaptive service to determine the time required to transfer the ghostzone data between machines. The application provides data on the required computation time. Using these two values, the adaptive component can specify to the application if the ghostzone size needs to be increased or decreased.

4.7 Communication Adaptation of Distributed Simulations (2)

This scenario extends the previous one with data compression for the exchanged ghostzones. Compression decreases the volume of data sent, but also increases the computation overhead. By estimating the compression ration that can be achieved, and the computational effort required, the adaptive component can determine if compression is useful.

4.8 Distributing Computation across Heterogeneous Processors

In this scenario, a parallel Cactus simulation simultaneously runs on processors of different speed. The problem is to distribute the application data (and with it the computational load) such that the overall simulation runs as fast as possible, balancing the load across the processors.
The load distribution needs to be estimated at program start and possibly during the run if severe load imbalance is detected, due to imprecise initial estimation, or due to the application dynamically changing its behavior (different simulation, analysis, I/O, etc.).
The adaptive component uses both static and historic information about the machines that are available and estimates the computational power of each machine and the entire system. It can then use this information to determine suitable data distribution. This distribution can be further refined by using application measurement.

4.9 Compute Time Estimation

The GRMS is supposed to schedule an application request such that the application completes within a given deadline (if at all possible). An alternative formulation of this problem is to determine the necessary resources in order to comply to a given deadline.
For this purpose, the scheduler needs information about predicted wait time in processor queues and about completion time of an application when scheduled to a given machine, or a combination of multiple machines, the latter in case of co-allocation.
This scenario can be implemented directly using the `estimate_usage` and `estimate_multiple_usage` calls in the adaptive service. These can be used to predict the behavior of any metric.

References

[1] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Coputer Systems*, 15(5–6):757–768, Oct. 1999.