Self-Managing Sensor-based Middleware for Performance Monitoring and Data Integration in Grids *

Hong-Linh Truong
Institute for Software Science,
University of Vienna
truong@par.univie.ac.at

Thomas Fahringer
Institute for Computer Science,
University of Innsbruck
tf@dps.uibk.ac.at

Abstract

This paper describes a sensor-based middleware for performance monitoring and data integration in the Grid that is capable of self-management. The middleware unifies both system and application monitoring in a single system, storing various types of monitoring and performance data in decentralized storages, and providing a uniform interface to access that data. We have developed event-driven and demand-driven sensors to support rule-based monitoring and data integration. Grid service-based operations and TCP-based data delivery are exploited to balance trade-offs between interoperability, flexibility and performance. Peer-to-peer features have been integrated into the middleware, enabling self-managing capabilities, supporting group-based and automatic data discovery, data query and subscription of performance and monitoring data.

1. Introduction

Grid monitoring is a crucial task as it provides performance and monitoring data for several functions such as performance analysis and tuning, performance prediction, fault detection, and scheduling. Grid monitoring middleware has to support monitoring of disparate resources and applications and to integrate performance and monitoring data from many sources.

To monitor various resources in Grids, a large number of monitoring sensors needs to be developed and deployed in different domains. In our view, such sensors are very similar to those in sensor networks [2, 30] in which the sensor follows resource constraints such as communication (e.g., sensors must use limited bandwidth), computation (e.g., sensors have to use limited computing power and memory sizes). These constraints limit data processing capability of a sensor thus normally the sensor sends collected data to a sensor manager. Also because resources on which sensors execute and resources sensors monitor may join and leave, the structure of sensor networks is frequently changed. Therefore, sensors and sensor managers must operate in self-managed, decentralized manner.

Most existing Grid monitoring tools have monitoring sensors operating in distributed manner and the network connecting sensors to sensor managers exploits the various types of communication such as shared memory [7], TCP [31], UDP [9], multicast [23]. However, these tools do not focus on the interoperability among sensor networks and the self-organization within them, and support limited types of sensors. Mostly they support event-driven sensors (e.g. in [7, 23, 31]). Sensor managers are configured into tree of point-to-point connections (e.g. in [7, 23]); or directory services, supporting discovery of data and sensor managers, do not interact with each other (e.g. in [31]).

Lack of interoperability among sensor networks and lack of self-organization within them have hindered distributed data discovery, data query and subscription (DQS) in Grid monitoring tools, not to mention fault-tolerance. Currently data discovery and DQS are mostly based on hierarchical or centralized models, as studied in [17]. But such models do not work well with more dynamic, large-scale distributed environments in which useful information services may not be known in advance. As suggested, e.g. in [27], and demonstrated, e.g. in [28, 19], the super-peer model and service group, powered by peer-to-peer (P2P) computing [24], have the advantages in solving the above-mentioned issues, but have not been exploited in Grid monitoring middleware. Moreover, most Grid monitoring tools are not capable of self-configuration and -reconfiguration under varying conditions which occur frequently in the Grid. Autonomic computing [21] which aims to cope with the unpredictable conditions of systems should be exploited.

Integrating performance and monitoring data in Grids is crucial because it is likely that no single tool will be deployed to provide performance data for all Grid sites and we

* This research is supported by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104.

need to utilize and analyze monitoring data across multiple Grids at the same time. Seamlessly integration and highly interoperability require well-defined interfaces, rich expressive customized data representations, and more power to process and store data. However, involving more function and processing results in slower performance. Therefore, we need to balance tradeoffs between interoperability and performance. Using Grid/Web service-based operations and XML data supports highly interoperability among different tools, easily customizing collected data, however, the performance considerably suffers when data is delivered via Web service operations with SOAP [14]. On the other hand, (parallel) TCP-based data streams can be utilized to achieve higher performance in delivering data in Grids [3]. Current Grid monitoring tools exploit either Grid service-based operations or TCP-based streams.

In this paper, we describe our first step in exploiting, developing and incorporating self-managing and P2P features into a sensor-based middleware for Grid monitoring and performance data integration within the SCALEA-G system [29]. The rest of this paper is organized as follows: Section 2 outlines our sensor-based architecture. Section 3 discusses our sensor model for performance monitoring and data integration. We then describe self-managing capabilities in Section 4. We discuss DQS in Section 5. Hybrid communication based on service-based operations and TCP-based streams are presented in Section 6. Section 7 illustrates experiments and examples. We present some related work in Section 8 before presenting our conclusions in Section 9.

2. Sensor-based Middleware Overview

Figure 1 depicts the architecture of sensor-based middleware implemented in SCALEA-G with the main Grid services named Directory Service, Sensor Manager Service, Client Service. These services, based on OGSA [15] and organized into service groups, support managing, storing and providing various types of performance and monitoring data measured and gathered by an extensive set of distributed sensors. They are capable of self-management and can collaborate in serving the requests from clients.

Directory Service (DS) stores information (e.g. schema, availability) about performance and monitoring data, SM and other services of the middleware. **Sensor Manager Service (SM)** manages sensors and data collected and gathered by sensors, and provides these data to consumers via DQS operations. An SM can interact with several sensor instances executed in distributed machines; sensor instances will send their collected data to SMs. SM uses XML containers to store performance and monitoring data. **Client Service (CS)** provides interfaces for administrating activities of SMs, querying data registered in DS, subscribing and/or querying data stored in SMs, etc.

SM and DS are organized into two types of groups (communities): *SM Group* and *DS group*. Within a Virtual Organization (VO) [16] there could be several SM groups. A DS group is deployed for multiple VOs; each VO provides a number of DSs which form the DS group. DSs register their information with a set of Registry Services. By using CS, the client of the monitoring middleware, exploring the monitoring service through existing Registry Services, can find DSs, SMs and then access performance and monitoring data. In our framework, we reuse existing implementation of Registry Service. However, DS and SM are specially designed for performance and monitoring purpose.

3. Sensor-based Model for Performance Monitoring and Data Integration

3.1. Monitoring Sensor Conceptualization

Sensors are used to capture performance data and to monitor resources including computational and network resources, and Grid applications. Every sensor monitors one or more *resources* (e.g. machine, network, Grid applications) and provides *measurement data* of the monitored resources; each resource is determined by a unique resource identifier (`ResourceID`) and measurement data is described in XML. Each sensor presents a *sensor profile* which describes the sensor, e.g. unique sensor identifier (`SensorID`), sensor description and lifetime, how to control the sensor (e.g. calling parameters) and information about the provided sensor data (e.g. XML schema of data). How a sensor works is described by the *sensor model*, e.g. event-driven or demand-driven, or rule-based monitoring. For measurement data, the tuple (`SensorID`, `ResourceID`) is unique that is used to determine monitoring data of a resource.

3.1.1. Event-driven and Demand-driven sensors Sensors in most existing Grid monitoring tools are based on event-driven model; a sensor measures and collects data based on events, mostly time-based event. Event-driven sensors collect the data and store the collected data when an event happens at a time, without consideration at that time the data is needed. Demand-driven sensors collect and provide data only when receiving requests. Demand-driven sensors are particularly useful for integrating data provided by other sources. To realize the importance of both types of sensors, our middleware supports both event-driven and demand-driven sensors.

3.1.2. System Sensors and Application Sensors We distinguish two types of sensors: *system sensors* and *application sensors*. System sensors are used to monitor and measure the performance of Grid computational services (e.g. computational hosts) and network services (e.g. network

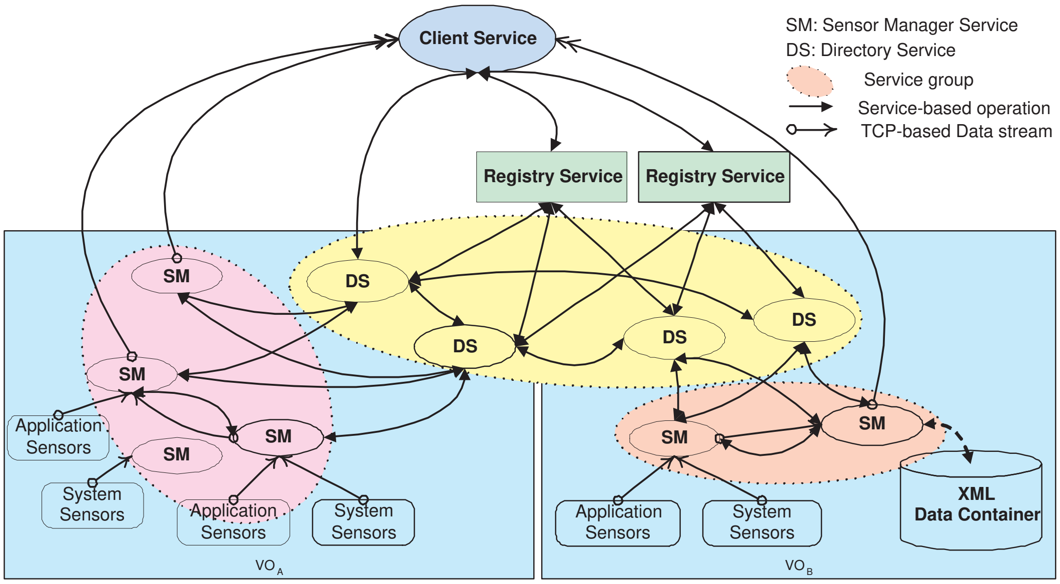

Figure 1. High-level view of self-managing sensor-based middleware.

```
Fuzzy bandwidth= new Fuzzy(0,5) {
 Shoulder VERYLOW =new Shoulder(0,1,ARL.Left);
 Triangle LOW =new Triangle(1,1.5,2);
 Trapezoid MEDIUM =new Trapezoid(2,2.2,2.8,3);
 Triangle HIGH = new Triangle(3,3.5,4);
 Shoulder VERYHIGH=new Shoulder(4,5,ARL.Right);
};
```

Figure 2. A fuzzy variable describing status of the bandwidth of a network path.

connections). Application sensors embedded in Grid applications are used to measure execution behavior of code regions and to monitor user-defined events in these applications. The two types of sensors are treated basically the same. They, however, differ in control and security model. This distinction allows us to simplify the management of two different types of sensors.

3.2. Rule-based Monitoring

Different from event-driven sensors in existing Grid monitoring tools, our event-driven sensors support rule-based monitoring. Instead of sending monitoring data it collects, the sensor uses rules to analyze monitoring data, and reacts with appropriate functions.

We use ABLE Rule Language (ARL)[11], which supports if-then-else rules, when-do pattern match rules, etc., to define rule sets for sensors. ABLE toolkit [10] provides a wide range of inference engines to process the ARL rule-sets, e.g. boolean forward/backward chaining, fuzzy forward chaining, pattern match engine. For example, to define a fuzzy variable for monitoring bandwidth of a network path in the Austrian Grid [4], we used Iperf [20] to test the bandwidth, and obtained the maximum observed bandwidth which never exceeds 5 MBytes/s. We divided the bandwidth into 5 states by using fuzzy logic as shown in Figure 2. Based on this fuzzy variable, we define a rule set, presented in Figure 3. With this rule set, depending on the status of bandwidth of the network path, e.g. *very low, low* or *very high*, the sensor will react with appropriate functions.

```
S1: bandwidth = getBandwidth();
R_VERYLOW: if (bandwidth is VERYLOW) {
          doReactionWhenBandwidthVeryLow();
         }
R_LOW:  if (bandwidth is LOW) {
           doReactionWhenBandwidthLow();
        }
R_VERYHIGH: if (bandwidth is VERYHIGH) {
         doReactionWhenBandwidthVeryHigh();
        }
R_OTHER:    doNormalReaction();
```

Figure 3. Example of rule set for bandwidth of a network path.

In the case where rules are not specified when a sensor is instantiated, the sensor instance will work as in the normal model (e.g. sending monitoring data when the event happens). Rule-based monitoring approach brings many advantages as it allows us to easily customize the monitoring actions. In addition, we can implement autonomic features that consider changing systems as an effect of the monitoring behavior. However, there is no common rule set for all resources even those monitored by a single sensor. Rules have to be built for each resources based on best practices.

3.3. Performance Data Integration by Using Demand-driven Sensor

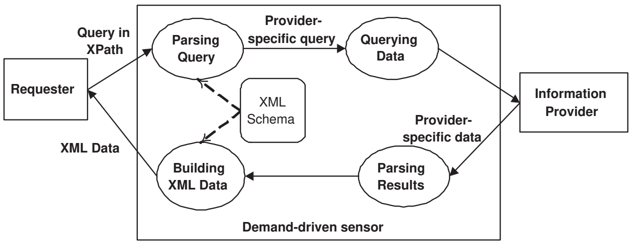

Figure 4. Using a demand-driven sensor to integrate performance data.

Besides using demand-driven sensors to monitor resources, we also exploit them for data integration. Figure 4 presents the model of using demand-driven sensor for integrating performance and monitoring data from other providers, e.g. MDS (Monitoring & Directory System) [12], NWS (Network Weather Service) [31], Ganglia [23]. To access different providers, we develop different demand-driven sensors taking the role of data mediators. As shown in Figure 4, when the sensor receives an XPath-based request from a requester, based on XML schema, it parses the request, extracting information of the request such as tag names with their associated attributes. The sensor then constructs a provider-specific request, calling the information provider with that request, and obtaining the result in provider-specific format. The sensor then parses this result and builds a new result described in XML. The XML-based result will be sent back to the requester. With this approach, other services use the same mechanism to access data in other providers as in our service.

When a demand-driven sensor is activated, the sensor returns information about resources whose monitoring data it can collect to the SM which in turn publishes the information to DSs. With that information, consumers can create requests for monitoring data. The requester only knows the XML schema of requested data in order to specify the request. The rest, where the requested data locates and how to get the requested data, are done by the middleware.

4. Self-Managing Services

4.1. Service Group

Each SM or DS group has a set of operations associated with the group. These operations address (i) how the requests for performance data are handled, and (ii) how the requested data are delivered. The real number of members of a group is dependent on the actual deployment which can dynamically change.

The group operations associated with SM group are group-based DQS. One member in the group can act as a mediator for other members. Given a DQS request, an SM can provide requested data even though its storage does not contain the requested data by collaborating with other SMs in the same SM group. For a DS group, the group-based operation supports the discovery of data providers. Given a request to find the provider of a needed data type, DSs in a DS group can cooperate in determining the data provider.

4.2. Data Dissemination and Maintenance

Instances of sensors are executed in monitored nodes and then send collected data to SM which in turn stores the data into its data container. SM automatically publishes characteristics of received data to a set of DSs, not to a single DS. Each SM keeps its group name and a list of DSs to which it publishes data. The list of DSs can be dynamically changed over the time. Each SM keeps a list of Registries where it can search information about DSs. When an SM is created, the SM gets a maximum number p of DSs it should register with and a pre-defined updated interval t seconds. In the cycle of t, SM looks up Registries to get n DSs. The SM then selects `min(n,p)` DSs from n ones. SM then disseminates information about data it stores to its selected DSs.

A DS can publish information about itself to multiple Registries. In the current implementation, one DS belongs to a DS group. A DS keeps a list of Registries with which it registers its information. A DS maintains a list of DSs in its groups; these DSs are its edge peers. Repeatedly with predefined t_r seconds, the DS searches Registries for up-to-date information about its edge peers. DS also performs `ping` test to its edge peers to check whether its edge peers are alive. To make sure that it provides the updated information, the DS checks its data in the database periodically based on a pre-defined t_d seconds. During the checking procedure, DS invokes `ping` operation of its registered SMs. If a `ping` to an SM failed, DS assumes that the SM is out of service and then DS removes all information associated with that SM. In the cycle of t_u seconds, DS publishes its information to Registries. Before DS finishes its execution, it unregisters its information from the Registries.

The availability of Registries, DSs and SMs is the key issue to fault-tolerance of the middleware. Instead of storing data into a centralized SM, collected data are stored over a set of distributed SMs, thus guaranteeing that a failure of one or many SMs does not bring the whole service down. Our SM publishes its information to multiple DSs, thus, not only data is widely disseminated and highly available but also it makes sure that if a DS is failed to serve requests from clients, still other DSs can do.

4.3. Discovery of Data Providers

Any client that wants to subscribe or query performance data of a resource has to locate a corresponding SM which provides the data. The discovery of data providers is based on requests containing tuples of (`SensorID`, `ResourceID`). A tuple (`SensorID`, `ResourceID`) is unique that determines monitoring data of a resource. Each DS provides a set of operations for other services to retrieve and search its registered data. With (`SensorID`, `ResourceID`), a client can invoke operations of a DS to discover data providers registered with that DS. (`SensorID`, `ResourceID`) can also be specified in data content filters of DQS requests.

Data discovery can also be done automatically by CS thus a client does not need to directly interact with DSs. CS parses client requests to get detailed elements such as `SensorID`, `ResourceID`. A list of DSs will be obtained from given Registries. The request is then sent to DSs which in turn cooperate in locating SMs by using group-based operations. When a DS cannot locate the provider of the requested data, it forwards the request to all its edge peers, otherwise it just sends back the results. These edge peers conduct the search and return the result to the peer who calls them. The DS then sends back the result to the requester. A parameter is used to control the request forwarding policy.

5. Data Query and Subscription

5.1. Query and Subscription Operations

SM provides a set of service operations for other services to subscribe and query data available in the SM, to unsubscribe and renew existing subscriptions. DQS requests consist of information about sensors and resources, data content filter, subscription duration (for subscription requests), etc. Content filters are described in XPath that can be easily written based on XML schemas of data provided by sensors. By using operations of SM, CS supports both *one-to-one* and *one-to-many* DQS requests. In one-to-one mode, a subscription or query request is used to obtain performance data provided by a single SM whereas in one-to-many mode a client subscribes or queries data from many SMs by using a single subscription or query request.

5.2. Automatic Query and Subscription

Clients can also perform DQS automatically without knowing where the requested data is located. The content filter, specified in DQS requests, can contain characteristics of data such as `SensorID`, `ResourceID`. When receiving a request from clients, CS processes the content filter and obtains `(SensorID, ResourceID)` information. It then searches DSs in order to find SMs that provide the requested data; the search is mentioned in Section 4.3. CS then sends requests to SMs which provides the requested data. As a result, the client does not necessary know where the monitoring data is stored. If DQS requests contain information about sensors and monitoring resources, the middleware can automatically handle DQS requests.

The middleware provides APIs for clients to conduct automatic DQS. The APIs hide all the lower-level details of the middleware. Figure 5 presents a simple code which is used to query available monitoring data of CPU usage of the machine `schareck.dps.uibk.ac.at`. The `ConsumerService` class is responsible for processing DQS tasks. The client knows a Registry Service and indicates the service handle of Registry Service (variable `handle`), specifies the content filter (variable `content_filter`), and calls the CS. The resulting data is retrieved through a `DataSensorReader`.

5.3. Group-based Data Query and Subscription

An SM can act as a mediator for other services to access data provided by other SMs in its group. When a client sends a DQS request to an SM, if the SM does not provide the requested data, SM will search its registered DSs to find SMs that can serve the request. If the search is successful, the SM acts as a super-peer between the data requester and the SM provider by forwarding the request to the SM provider. The provider first tries to communicate with the requester. If successful, the provider sends requested data to requester, otherwise it sends data back to the caller. If an SM receives request from another SM, it will not propagate the request when it cannot serve the request. In this model, an SM can take the role of the super-peer in either/both forwarding requests or/and delivering data. Any SM may become a super peer at the runtime.

5.4. Notification

In all mentioned DQS, the client conducts DQS based on available information about monitoring data published

```
ConsumerService cs = new ConsumerService();
cs.activateUpDataService();
String handle="http://bridge.vcpc.univie.ac.at:8765/ogsa/services/
                               samples/registry/VORegistryService";
String content_filter="/sensordata[@SensorID=\"host.cpu.used\"]
                        [@ResourceID=\"schareck.dps.uibk.ac.at\"]";
SensorDataReader out =cs.distributedQueryDataWithRegistry(handle, content_filter);
```

Figure 5. Example of querying monitoring data by using information from Registry Service.

in DS. However, there are many cases in which the client wishes to subscribe for a notification of interesting data which is not available at the time of the subscription. For example, the client may inform the monitoring system that it wishes to receive execution status of activities of a workflow application being executed by the workflow enactment before it submits the workflow application. We call this type of data subscription *notification subscription*.

DS and SM provide two service operations named `subscribeNotification`, `unsubscribeNotification` for subscribing and unsubscribing notification data. DS and SM use a table to keep existing subscriptions of notifications. The client can subscribe the notification on a specific SM or on the whole monitoring system. If the client wishes to receive notification message from a specific SM, the client can register with the SM by calling `subscribeNotification` operation of that SM. In this case, the client will not receive notification data collected by other SMs even though that data satisfies the client's request.

In our framework, SM gathers and stores performance data collected from sensors. However, there is no mechanism to determine SMs which are capable of distributing a specific notification data because SMs and sensors can enter and exit the monitoring system arbitrarily. The client may only know of a few services to which it contacts, e.g. a DS or an SM, but it wishes to receive a notification without knowing the service which is capable of providing this notification. We support this type of notification subscription by implementing a global notification mechanism. By using the CS, the client registers with a set of DSs $\{DS_1, DS_2, \cdots, DS_n\}$ that it knows, and indicates information about interesting data which it wishes to be notified. Each DS_i updates the table containing subscriptions of notifications and then calls `subscribeNotification` operation of registered $\{SM_{i1}, SM_{i2}, \cdots, SM_{im}\}$ in its directory with that indicated information. Similarly, when a new SM registers with a DS, the DS calls that operation of the SM with existing subscriptions in its table. When receiving a `subscribeNotification` call, the SM updates a table containing tuples of `(ResultID, Subscription)`. Whenever SM receives data satisfying notification constraint, SM delivers the data to CS. If SM cannot deliver a notification to a client, the SM will remove the subscription of that notification from the table. To unsubscribe a notification, CS sends unsubscription requests to DSs which in turn pass these requests to SMs.

6. Service-based Operations and TCP-based Data Delivery

Each SM can be viewed as a peer in a P2P network. It, however, also is a Grid service. In most P2P systems, a peer processes request and delivers data via TCP/UDP channels. Our peer is unique as we try to integrate both concepts, P2P model and Grid service, into a single peer. A peer provides Grid service operations for other peers and high-level clients to access and control its service. However, peers use TCP-based streams to deliver monitoring data, thus data delivery among peers can be easily implemented and it gives a higher performance for data delivery.

Figure 6 depicts how requests for data and the requested data are handled. CS or SM requests data through *Grid service-based invocations* whereas requested data is delivered via *TCP-based streams*. *Data Sender, Data Receiver* and *Data Relay* of SM and CS are responsible for sending, receiving, and relaying performance data, respectively. An SM has only one connection to a consumer for delivering all kinds of subscribed data. The connection is created at the first subscription and will be freed after pre-defined t_δ seconds since the last subscription finishes. For delivering resulting data of queries, an on-demand connection will be created and freed when the delivery finishes. A request for data always specifies a unique `ResultID` which is associated with requested data satisfying the requested constraints. SM uses `ResultID` to route requested data to the destination while CS uses `ResultID` to aggregate results of the same request delivered from multiple SMs.

In our middleware, monitoring data is described in XML. While using XML to describe performance and monitoring data provides a widely accessible interface and simplifies the interoperability among services, XML data grows in size. To reduce the size of monitoring data transfered, we compress the data before sending it over the network.

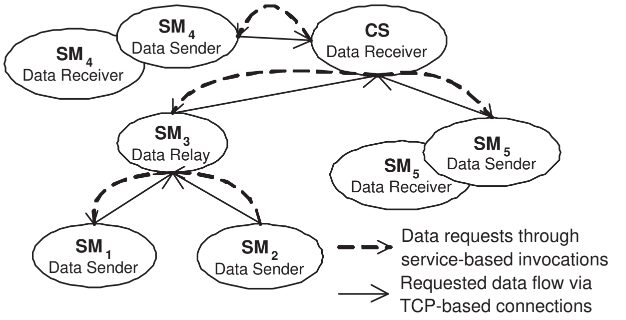

Figure 6. Service-based invocations and TCP-based data streams.

In Table 1, we monitored the data size and transfer time of CPU usage data from an SM in UIBK domain to a client in PAR domain (see Section 7 for more detail about the experimental test bed); transfer time is the average value of observed values at different times. In our measurement, compression ratio, compression and decompression time are all dependent on the data size and the type of monitoring data. When the data size is large, compressing data reduces the data size substantially which improves both data transfer time and throughput significantly. For most types of monitoring data supported, when the size of data to be transfered is less than 512 bytes, the compression does not achieve a better transfer time and throughput because the compression ratio is close to 1. Therefore, we develop a simple self-adaptive mechanism for deciding whether the resulting data should be compressed before sending to the requester that is based on the size of delivered data. If the data size is larger than s_δ, the data will be compressed, otherwise data is transfered as normal. Currently, s_δ is set to 512 bytes.

Data size (bytes)	T_f (ms)	T_{fcd} (ms)	r	$T_c + T_d$ (ms)
588	110	109	1.863	2
1560	119.33	115.24	4.537	2.11
3019	120.45	114.85	8.137	2.19
3991	120.68	114.68	10.207	2.44
5449	129.2	116.15	13.257	2.53
6421	130.45	116.05	14.967	2.73
7879	131.63	117.85	17.316	2.76
8851	136.28	117.48	18.712	3.08
10309	141.39	117.68	20.742	3.09
11281	143.25	117.93	21.949	3.22
12739	147.83	117.5	23.548	3.42
13711	149.75	117.25	24.355	3.67

Table 1. Example of transfer time without compression (T_f), transfer time of compressed data (T_{fcd}), compression ratio (r), compression and decompression time ($T_c + T_d$) for CPU usage data.

7. Experiments

Our middleware is implemented based on GT 3.2 [18], Java Cogkit [22], with various other libraries. We have deployed our sensor-based monitoring infrastructure on three domains: VCPC (University of Vienna), UIBK (University of Innsbruck) and GUP (Linz University) in the Austrian Grid [4]. Figure 7 presents our experimental test-bed. We set up three SM groups named SM-VCPC, SM-UIBK and SM-GUP in VCPC, UIBK, GUP, respectively. We establish a DS group that includes one DS in VCPC (DS-VCPC) and one in UIBK (DS-UIBK). Each DS stores data in a PostgreSQL database server which can be executed on the same domain (e.g. in case of DS-UIBK) or different one (e.g. DS-VCPC). SM stores collected data into XML containers implemented atop Berkeley DB XML [1]. There are two Registries in VCPC and UIBK. A client is deployed in PAR domain (in University of Vienna). All DSs and Registries can be accessed by all SMs and clients, but only SMs executed on `bridge/VCPC`, `olperer/UIBK`, `iris/GUP` can directly deliver data to the client executed on `kim/PAR`.

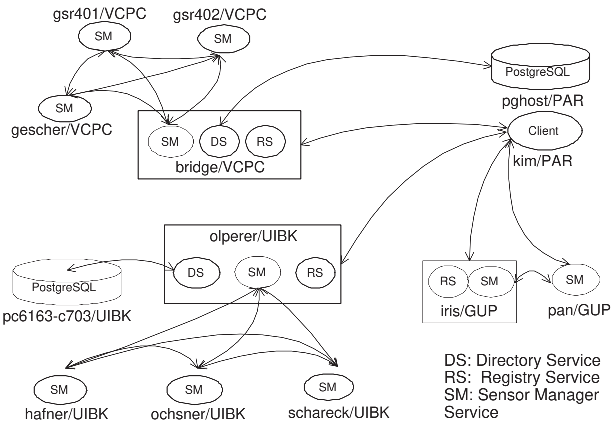

Figure 7. Experimental test-bed.

7.1. Performance Analysis of Data Discovery

To evaluate the performance of the discovery of data providers within the test-bed, we setup two modes. In *one-to-one mode*, a client sends requests directly to DS which in turns finds data providers of the requested data. If the DS cannot locate the data provider, it will not send the request to other DSs in the same group. In *group mode*, if the DS cannot answer the request, it sends the request to its edge peers in its group asking for the location of data providers. In both modes, a client in PAR domain sends requests to DSs at the same time.

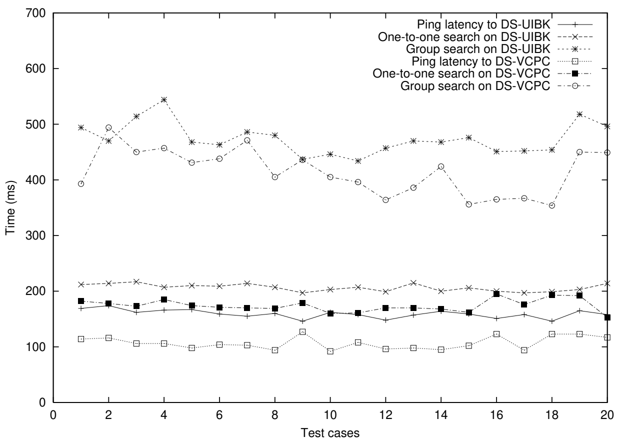

Figure 8. Ping latency and search time.

Figure 8 presents search time in one-to-one and group mode, and latency of `ping` operation. Overall for both DSs, the ping latency is larger than a half of the time spent on the search for data providers in one-to-one mode, suggesting that the time DS spends in searching its database is small when compared with ping latency. A considerable portion of time spent in the discovery process is service-operation latency from client to service. In our implementation, for group mode, DS creates a new thread which calls an edge peer when the DS cannot locate the data provider. Search time in group mode nearly doubles that in one-to-one mode partially because at the same time a DS has to fulfill a request from an edge DS and to forward a request to its edge DS. The latency from client domain (PAR) to DS-UIBK is higher than that to DS-VCPC, also DS-VCPC is executed on an SMP machine where DS-UIBK is executed on a single CPU machine. Therefore, conducting group-based and one-to-one discovery through DS-VCPC is considerably faster than that via DS-UIBK due to the differences of network latency and computation power.

7.2. Monitoring and Data Integration Example

Figure 9 presents an analysis of profiling data collected by application sensors. Online profiling data of Grid applications is incrementally sent to SMs. The application profile analyzer then conducts DQS on profiling data, analyzing and visualizing the results to the user. The left window of Figure 9 shows code regions associated with their processing units (compute node, process, thread). For each code region, profiling metrics are displayed in the right window.

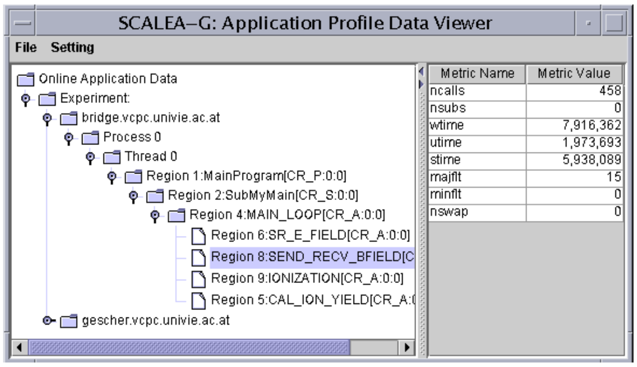

Figure 9. Analysis of application profiling data.

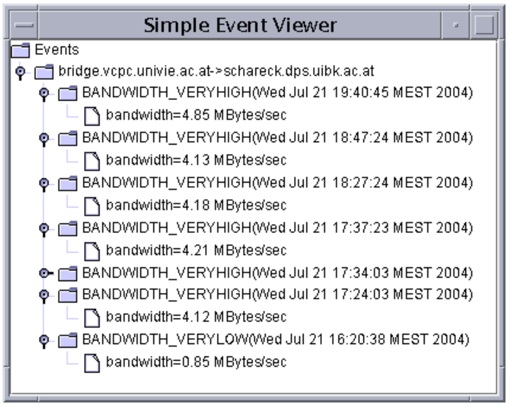

Figure 10. Events generated by a rule-based sensor monitoring network bandwidth.

Figure 10 presents an example of bandwidth monitoring of a network path from VCPC to UIBK. We setup a simple rule set based on fuzzy variable for the bandwidth, as presented in Section 3.2. Only when the bandwidth of the network path is *very low, low* and *very high*, the sensor sends events to SM. Events are subscribed and visualized by a simple generic event viewer as shown in Figure 10.

Figure 11 presents few snapshots of monitoring system load, CPU usage and network delay roundtrip (monitoring data provided by event-driven sensors), and of forecasting CPU usage and TCP bandwidth (forecasted data provided

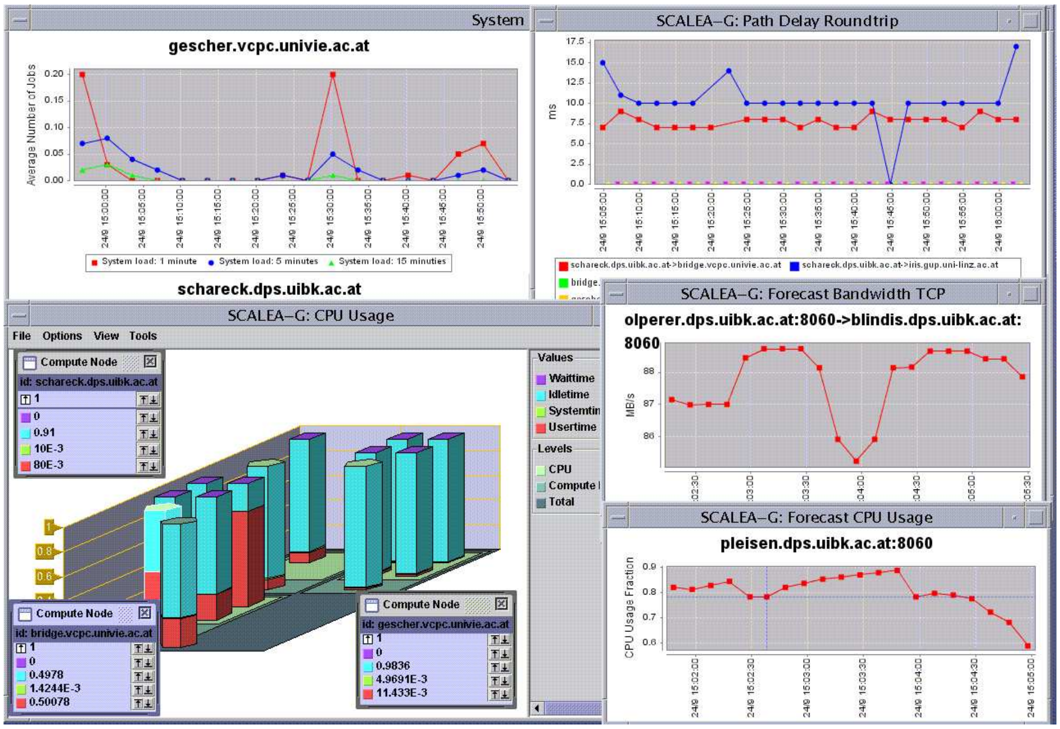

Figure 11. Snapshots of online monitoring system load, CPU usage and networks.

by demand-driven sensors). For example, data about CPU usage (waiting time, idle time, system time, user time) are measured per CPU. CPU monitoring data is periodically collected and the change of CPU usage can be observed on the fly through data subscription (see window *CPU Usage*).

8. Related Work

Over the past few years, many Grid monitoring and performance tools have been developed, as cataloged in [17]. Several existing tools are available for monitoring Grid computing resources and networks, e.g. MDS [12], NWS [31], GridRM [6], Gangila [23] and many of them are based on the generic Grid Monitoring Architecture (GMA) model [5]. However, few tools have been developed for monitoring Grid applications. e.g., GRM [25], OCM-G [8].

None of aforementioned systems, except MDS, is OGSA-based service. They support the monitoring of either infrastructures or applications while we unify both in a single system. Most existing tools employ communication either based on TCP-based streams or based on Web/Grid service whereas our middleware exploits both Grid service invocations and TCP-based data streams. These tools, although conduct distributed monitoring, mostly support data discovery and DQS based on hierarchical and centralized models, and event-driven sensors without rule-based monitoring. We use decentralized data storages, and support event- and demand-driven sensors, and rule-based monitoring.

The use of actuators to enable and configure resource management, e.g. in [26], is one aspect of using monitoring data for self-configuring. Based on rule sets, our sensors can self-manage its actions in processing monitoring data of monitored resources but we do not provide actuators/effectors that control monitored resources yet.

9. Conclusion and Future Work

Due to the diversity and dynamics of the Grid, monitoring middleware needs to unify and provide different types of monitoring sensors such as system and application sensors, event- and demand-driven sensor, and to integrate various types of data from many sources. Exploiting both Grid service-based operation and TCP-based data stream can help balancing tradeoffs among interoperability, manageability and performance. Middleware must store collected data on distributed sites, providing the same mechanism for accessing that distributed data. By incorporating P2P and autonomic technologies, Grid monitoring middleware is capable of self-organization, supporting group-based data discovery and DQS. As a result, it helps increasing availability and reliability of the middleware as well as dealing with the dynamics of large distributed environments. This paper contributes on the design and implementation of a Grid monitoring middleware that exploits the above-mentioned points.

We are currently improving our prototype and investigating to port our framework to WSRF [13]. Although P2P and autonomic features give many promising solutions to solve challenges in Grid monitoring, it is not a simple task to incorporate these features into a Grid service-based middleware. To continue our effort on utilizing sensor networks, P2P and autonomic computing features, the set of sensors will be extended, together with effectors, to support self-healing. We plan to provide adaptive capabilities for SM and DS so that they can self-adjust their functions under the computing capabilities of the hosting environment.

References

[1] Sleepcat Berkeley DB, http://www.sleepycat.com.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, Aug 2002.

[3] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Comput.*, 28(5):749–771, 2002.

[4] AustrianGrid. http://www.austriangrid.at/.

[5] B. Tierney et. al. A Grid Monitoring Architecture. Technical report, Performance Working Group, Grid Forum, January 2002. http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf.

[6] M. Baker and G. Smith. GridRM: An Extensible Resource Management System. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER'03)*, pages 207–215, Hong Kong, December 01-04 2003. IEEE Computer Society Press.

[7] Z. Balaton and G. Gombas. Resource and Job Monitoring in the Grid. In *Proceedings. Euro-Par 2003 Parallel Processings*, Klagenfurt, Austria, 2003.

[8] B. Balis, M. Bubak, W. Funika, T. Szepieniec, and R. Wismüller. An infrastructure for Grid application monitoring. *LNCS*, 2474:41–49, 2002.

[9] O. Barring. Towards automation of computing fabrics using tools from the fabric management workpackage of the eu datagrid project. *ECONF*, C0303241:MODT004, 2003.

[10] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[11] I. T. W. R. Center. ABLE Rule Language: User's Guide and Reference. Version 2.1.0.

[12] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[13] I. F. el al. Modeling Stateful Resources with Web Services. Specification, Globus Alliance, Argonne National Laboratory, IBM, USC ISI, Hewle tt-Packard, Jan. 2004.

[14] R. Elfwing, U. Paulsson, and L. Lundberg. Performance of SOAP in Web Service Environment Compared to CORBA. In *Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 84–, Gold Coast, Australia, December 04 - 06. IEEE Computer Society.

[15] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, pages 37–46, June 2002.

[16] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[17] M. Gerndt, R. Wismueller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. *Performance Tools for the Grid: State of the Art and Future*, volume 30 of *Research Report Series, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen*. Shaker Verlag, 2004. ISBN 3-8322-2413-0.

[18] Globus Project. http://www.globus.org.

[19] J. Hwang and P. Aravamudham. Middleware Services for P2P Computing in Wireless Grid Networks. *IEEE Internet Computing*, 8(4), 2004.

[20] Iperf. http://dast.nlanr.net/projects/iperf/.

[21] J. O. Kephart and D. M. Chess. Cover feature: The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[22] G. Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(643-662), 2001.

[23] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, May 2004.

[24] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-To-Peer Computing. Technical Report HPL-2002-57, HP Labs, March 2002.

[25] N. Podhorszki and P. Kacsuk. Design and implementation of a distributed monitor for semi-on-line monitoring of visualmp applications. In *DAPSYS'2000 Distributed and Parallel System, From Instruction Parallelism to Cluster Computing*, pages 23–32, Balatonfured,Hungary, 2000.

[26] D. A. Reed, H. Simitci, and Y. L. Ribler. Autopilot performance-directed adaptive control system. Jan. 10 1998.

[27] D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(04):96–95, 2003.

[28] D. Talia and P. Trunfio. Web Services for Peer-to-Peer Resource Discovery on the Grid. In *DELOS Workshop: Digital Library Architectures*, S. Margherita di Pula, Cagliari, Italy, 24-25, June 2004. Edizioni Libreria Progetto, Padova.

[29] H.-L. Truong and T. Fahringer. SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. *Scientific Programming*, 12(4):225–237, 2004. IOS Press.

[30] M. Tubaishat and S. Madria. Sensor networks: an overview. *IEEE Potentials*, 22(2):20–23, April-May 2003.

[31] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems*, 15:757–768, 1999.