



Project No. FP6-004265

CoreGRID

European Research Network on Foundations, Software Infrastructures and Applications
for large scale distributed, GRID and Peer-to-Peer Technologies

Network of Excellence

GRID-based Systems for solving complex problems

D.IM.02 – Proposal of architecture for scalable GRID monitoring architecture

Due date of deliverable: August 31, 2005

Actual submission date: October 10, 2005

Start date of project: 1 September 2004

Duration: 48 months

Organisation name of lead contractor for this deliverable: MTA SZTAKI

Revision: *Final*

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: grid, monitoring systems, architecture

Contents

1	Introduction	2
2	Related work and existing systems	2
2.1	Grid Monitoring Architecture (GMA)	2
2.2	Mercury Monitoring System	3
2.3	R-GMA	3
2.4	Other lightweight implementations	3
2.5	Related standards and protocols	4
3	Requirements and current problems	4
3.1	Resource monitoring	4
3.2	Application and job monitoring	5
3.3	Service monitoring	7
3.4	Workflows	8
3.5	Consumer-side requirements	8
3.6	Interoperability	8
4	Proposal for a generic GRID monitoring architecture	9
4.1	System components	9
4.2	Data model	10
4.3	Component interactions	11
4.4	Capability and attribute management	12
4.5	Layered system architecture	14
5	Proposed capability languages	16
5.1	Condor ClassAds	16
5.2	XML XPath based capability language	17
6	Considered data models	19
6.1	Trivial relational language	19
6.2	Interface to the Mercury Monitoring System	21
7	Plans for the prototype implementation and future research	22
8	Conclusion	23
A	Complete list of operations for C-GMA components	26
A.1	Functions supported by the producer	26
A.2	Functions supported by the consumer	26
A.3	Functions supported by the registry	27
A.4	Functions supported by the mediator	27

1 Introduction

Gathering information about the state and performance of a large scale distributed system is essential for system management, failure detection, accounting, auditing, performance tuning and intelligent steering. Providing such information however is not easy. There were several monitoring systems created in the past (some are described in section 2) that cover more or less aspects of grid monitoring but there is still no generally accepted architecture for unifying the existing approaches.

This document defines a generic architecture that addresses the monitoring requirements arising in grid environments. The goals of this architecture definition are twofold: first, it provides an infrastructure for utilising different existing monitoring system implementations within the same infrastructure. The second goal is to provide support for middleware developers who need to design and implement new monitoring solutions to suit their specific requirements.

This document is organised as follows. Section 2 describes existing monitoring systems and research work. Section 3 lists common monitoring scenarios as use cases and describes known problems and monitoring requirements for these scenarios. Section 4 describes the *Capability-based Grid Monitoring Architecture* (C-GMA) architecture at an abstract level while sections 5 and 6 give a description of a two-layered interpretation of the generic architecture. Section 5 proposes two capability languages and section 6 describes how some existing monitoring systems can be hooked into C-GMA. Finally section 7 gives information about the expected prototype implementation and future research directions.

2 Related work and existing systems

In this section we discuss some existing monitoring systems and related work without the incentive of being complete. For a wider coverage of monitoring solutions and comparison based on various criteria see [1] and [2].

2.1 Grid Monitoring Architecture (GMA)

The Grid Monitoring Architecture (GMA) [3] is an abstraction of the essential characteristics needed for scalable high performance monitoring on a large distributed computational Grid. The GMA proposal [3], published by the GGF Performance Working Group, defines the components producer, consumer and directory service. Besides component definition, proposal for interaction between components and required operations for each components are also defined. Basic requirements on monitoring architecture are defined as low latency, capable of high data rate, minimal overhead, security and scalability.

In order to separate data discovery and data transfer, a service called "directory service" was defined. Metadata describing producers of information and consumers waiting for some information are published in this service. Producers and consumers can then use published information to locate corresponding parties.

Several communication models between producers and consumers were identified: publish/subscribe, query/response, and notification. Operations supporting all types of communication are defined for producer and consumer. Basic operations for directory service were also identified. In all communication models, monitoring data and control messages are sent directly between a particular consumer/producer pair and directory service is involved only in location of corresponding sink.

In GMA, monitoring data are sent in the form of (time-stamped) events. Events are sent only from producer to consumer, but communication can be initiated by both parties.

Compound components, which implements both producer and consumer interfaces are also studied.

Interoperability of different monitoring systems was one of motivations for the GMA. Definition of the GMA inspired several implementations of monitoring systems. While the GMA definition proved to be general enough to cover all these implementations, it proved to be too much general to guarantee real interoperability between proposed architectures. The GMA definition does not constrain communication protocols between components nor data model for description of producers and consumers.

2.2 Mercury Monitoring System

The Mercury Monitoring System [4, 5] developed in the GridLab [6] project is a generic GMA-compliant grid monitoring system. It is designed to support both resource and application monitoring. Mercury also supports *actuators*. Mercury has a modular infrastructure where individual sensors and actuators are implemented as loadable modules thus providing easy extensibility. Mercury uses a compact binary protocol to reduce network traffic and communication overhead.

Mercury provides an instrumentation library that applications can use to publish their own internal data like performance counters or application events. Applications can also register controls that can be invoked through the monitoring system therefore providing remote steering capability.

Mercury also features a hierarchical design where lower level monitoring components are aggregated under higher level components. Mercury supports GSI authentication, data encryption and access control lists to address security concerns. Mercury is implemented using the C language but the producer and the consumer API are also available in Java.

2.3 R-GMA

Relational Grid Monitoring Architecture (R-GMA) [7] developed in the European Data-Grid project is a relational database implementation of the GMA. R-GMA is planned to be used not only as monitoring infrastructure, but also as generic information service. The R-GMA system uses web servlets implemented in Java, with SQL-like API. Client-side APIs are available in Java, C, C++, Perl and Python. According GMA implementation comparison ([8]), R-GMA has several drawbacks – large overhead both in run-time and during compilation and installation. From the developer's standpoint, rapid development and changing nature of R-GMA is also problematic. R-GMA API is also much more complex, according to technical report [8] it contains more than 200 API calls, while jGMA or pyGMA APIs have 46 or 17 API calls.

2.4 Other lightweight implementations

jGMA [9] is a lightweight implementation of the GMA in Java. It contains a very reduced API, no support for web-services or SSL/GSI authentication. It has a registry API that allows associating an XML description of features and capabilities with the registered entities. The jGMA supports different communication models for local and wide-area networks. The jGMA implementation is available only as a binary release.

pyGMA [10] is an lightweight implementation of the GMA in Python. It has web-service SOAP interfaces. The implementation contains just a framework, only some sample producers and consumers are provided. The distribution also contains a simple central registry.

CODE [11] is a Java based implementation of the GMA. CODE defines the components directory service, observer (provides information), actor (which can be asked to perform actions) and manager. Manager asks observers for information, reasons upon that information, and asks actors take actions if needed. Implementation includes support for GSI. An LDAP server is used for directory service. Several grid services were implemented using CODE, however the framework is currently not supported.

2.5 Related standards and protocols

Currently there is no commonly accepted protocol for interacting with a monitoring system.

The emerging web-services standards correspond to the proposed communication model between producers, consumers, registry and mediator. Particularly the WS-Notification draft [12] fits the component interaction model described in section 4.3. Parts of the WS-Resource Framework (WSRF, [13]), especially WS-ResourceProperties (WSRF-RP) corresponds with the capabilities described in section 4.4.

The OGSA-WG working group in the Global Grid Forum (GGF, [14]) has plans to define the OGSA Information and Monitoring Architecture based on the OGSA Architecture document [15] and the GMA proposal [3].

2.5.1 Capability languages

In the context of grids different languages for description of component capabilities were studied: classified advertisement (ClassAd, [16] for bi-lateral matching, set-based extensions of ClassAds [17] for specifying and solving set-matching problems, gang matching [18] for allowing requests to specify a list of bi-lateral matches, or the constraint-based language Redline [19]. See [19] for a survey in this research area.

The Grid Resource Allocation Agreement Protocol Working Group (GRAAP-WG) in the Global Grid Forum is working on the WS-Agreement protocol, which supports the establishment and management of agreement-based service relationships for both a service provider and a service consumer.

3 Requirements and current problems

In this section we have collected typical usage scenarios where monitoring is needed on the grid. These scenarios are used to demonstrate the need for specific capabilities of the monitoring system and therefore justify design decisions made further in this document. There are also descriptions of some problems that current monitoring systems do not address well.

Some of the requirements collected here will be addressed by the generic architecture described in section 4 while others are up to the actual implementation to fulfil.

3.1 Resource monitoring

Resource monitoring is probably the oldest and most well-known area of monitoring. Under the term *resource* we understand the physical building blocks of a grid, namely hosts, networks and users. Users are somewhat special because while they are in some sense very real they do not communicate with other grid components directly but use software components to represent them thus having a virtual presence only.

Resources may be organised into higher level entities (like clusters or user groups). In such a case it is desirable from a monitoring system to be able to provide aggregated information about these high level entities rather than just about the individual components.

Monitoring of hosts typically involve running some software component on them since most information (like CPU utilisation, system temperature or disk usage) can only be measured locally. This local component may be provided by the system software (under this term we understand every locally installed software that has no understanding of the grid, not just the bare operating system) or by the grid monitoring system itself.

Network monitoring is a large topic in itself which we do not want to discuss here in detail. The only important aspect that should be mentioned here is that although many network parameters are best measured at the various network equipment (routers, switches etc.) grid middleware usually has no access to these equipment therefore it has to resort to perform measurements between network endpoints yielding less reliable results. Also,

interpreting the results may require knowledge about the physical network topology which again is not always available to the grid middleware.

Other special hardware components (like storage arrays) are either connected to a specific host thus falling into the category of host monitoring or (like networks) can be monitored only from nearby hosts using indirect measurements.

User monitoring is not so well understood in grid context as host or resource monitoring is. This is due to the fact that users are not physically present in the system but are represented only by various software components acting on their behalf. However, both accounting (determining the amount of resources a user has consumed) and auditing (determining what kind of actions an user has taken on the grid) rely on user monitoring. Data that can be collected about grid users include aggregated resource usage, activity records and capability information.

One example of user monitoring that is not addressed well by existing systems is a job that requires interactivity. Current grid systems only support interactive jobs by trying to start them immediately and returning an error if that is not possible. A better approach would be to schedule the job as normal (probably with some time constraints, like it should be started only during work hours). Before really starting the job the broker could then check if the user is on-line at the moment, and postpone the job if the user is not available.

Resource monitoring yields two main classes of data: current state information and event reporting (e. g. failure notifications). State information is best suited to be delivered using the *pull* model meaning that an entity interested in the data must actively ask for it. Event information however is best delivered using the *push* model where the component that generated the event arranges for a notification to be sent to all interested parties. Therefore it is desirable for a monitoring service to support both push and pull data delivery models.

The two delivery models however can supplement each other to some degree. Delivering state information using the push model either means wasting resources (sending out data when nobody is interested in it) or reduced accuracy (when the delay between two consecutive events are bigger than what the receiving entity would require). Delivering event notifications using the pull model may result in losing messages if they arrive with a high frequency or introducing unnecessary delay for important messages.

In the case of resource monitoring both state and event information have usually a relatively small amount of data and a low rate. There are some areas having different requirements though. For example, Intrusion Detection Systems may generate events with a very high rate.

Summarising the requirements for resource monitoring we get:

- Both push and pull delivery model are useful.
- There may be important events that must be delivered with very low latency.
- There may be data sources generating events with a high rate thus throughput can be also important.
- There are a lot of information that can be collected and it would be very hard to create a single system covering everything. Thus being able to integrate new data sources into the existing system and being able to incorporate already existing measurement tools are important.

3.2 Application and job monitoring

Work units submitted to a grid system are called *jobs*. Jobs may describe computational tasks, data transfer requests or other activities.

The monitoring of computational jobs means two different things: monitoring the state of a job, or monitoring the internals of a running job. We will use the term *job monitoring* for the former and the term *application monitoring* for the latter.

Job state monitoring usually requires interacting with local and grid-wide resource managers and thus can be considered a form of service monitoring. Usually the application itself is not involved in job state monitoring but in some cases it may be instrumented to provide more timely or more fine-grained information than the resource manager is able or willing to give.

Following the life cycle of jobs is very important in production environments therefore the monitoring system providing job state information should be very robust against failures. This usually implies some form of persistence (storing the data locally before sending it to remote consumers) and the ability to deduce the job state even if some events have been lost. These techniques are all utilised by the Logging and Bookkeeping service used in the European DataGrid and EGEE projects.

Monitoring the internals of a running application requires the application itself to actively communicate with the monitoring system. This is usually achieved by instrumenting the application (inserting special code fragments into the application's code to generate various events). The instrumentation may be done manually by the application developer or performed by some tool automatically.

Instrumented applications may produce very large amounts of events. Monitoring systems targeted for application monitoring thus must either be able to deliver events efficiently and with a very low overhead, or they must support sophisticated data reduction techniques like OCM-G [20].

One common problem with application monitoring is identifying the grid job a given application process belongs to [5]. This is caused by the fact that grid sites use various local resource management systems internally that are not aware of grid jobs. Such local resource management systems often do not have external interfaces that a monitoring system could use to discover the identity and the real owner of an application process.

Intelligent steering of complex applications (especially distributed ones) requires not only receiving information about the application but also being able to alter their execution if there are changes in the grid or user environment. Therefore supporting the propagation of events not only from the application to the user but also in the reverse direction is a natural extension of a monitoring system.

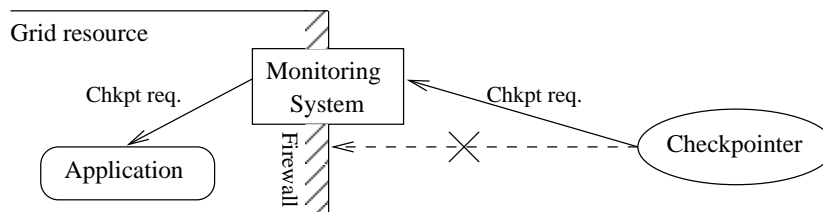


Figure 1: The monitoring system can deliver the checkpoint request when direct communication with the application is not possible

A special case of application steering is delivering *checkpoint requests* to the application. Using the monitoring system for this purpose has several advantages: first, grid resources often do not allow direct access to the computing nodes where the applications are running from outside thus contacting the application directly is not possible (see figure 1). Also, a monitoring system that supports application monitoring can already communicate with the application and reusing this communication channel is simpler than creating a different service, both for the application developer and the grid middleware provider.

Application monitoring imposes another requirement for the monitoring system: applications should be able to register their own information sources and control/steering capabilities that are only valid while the application is running. This means that the monitoring system must support dynamic schema changes, and the users of the monitoring

system must be able to discover these schema changes.

Summarising the requirements for job monitoring:

- Efficient support of job monitoring requires cooperation from both the grid-level and local resource management systems.
- Providing an instrumentation library in order to applications being able to define their own events is desirable. This also implies the support for dynamic schema changes.
- The monitoring system should be able to handle the large amount of data generated by instrumented applications.
- Supporting distributed data reduction alleviates the need for transmitting a large amount of instrumentation data.
- The monitoring system should utilise persistence for precious data and be robust against component failures and data loss.

3.3 Service monitoring

Service monitoring has two aspects: monitoring the availability of a service, and monitoring the internal workings of the service. The former can be achieved by either asking the service about its own status if it supports such a request, or by trying to invoke a predefined set of probing operations and watching for unexpected results.

Monitoring service internals means instrumenting the service itself to send important events to the monitoring system. This is essential for large, complex grid infrastructures where gathering information fragments from local log files spread across several locations is hard at best and impossible at worst. Such a service instrumentation technique was used on the GridLab [6] testbed to implement a grid-wide logging service.

Grid services are often not bound to a specific location and the monitoring system must be able to accommodate this. This means that services should be identified by some mechanism that is not affected when the location of a service changes. An interesting problem is whether a service restarted at a different location is really a continuation of the old service or should be considered a different one. This may be important for the users of the service but can only be decided on a case-by-case basis.

Being able to control services through the monitoring system, similar to application steering described in the previous section, is also desirable. One can argue that services already provide public interfaces so there is no need for supporting service control through the monitoring system. However it can still be justified since the monitoring system controls are to be used mainly for service management and internal tuning. Therefore the functionality provided through the service's own interfaces is orthogonal to the functionality provided by the monitoring system controls.

On the other hand if a grid infrastructure is built on top of WS-RF [13] compliant services then it may be desirable to define a common monitoring port-type. Through this port-type every service could provide some basic status information like the service is alive or where to get more service-specific information. This would allow monitoring systems to retrieve service status in a uniform way.

Requirement summary coming from the monitoring of services:

- There should be a way to uniquely identify monitored entities. In the case of services, this identifier should not depend on the current location of the service.
- Administering and controlling a service is orthogonal to the normal functionality provided by the service therefore it is justified to utilise the monitoring system instead of integrating steering capabilities directly into the service's standard interface.

- On the other hand having a standardised way to access service status information would reduce the complexity of the monitoring system and would help interoperability.

3.4 Workflows

Workflows are an important area in the CoreGrid project and thus their requirements for monitoring should be discussed.

Application workflows consist of several jobs having some dependencies between them. Monitoring of a single job that is part of some workflow is already covered in section 3.2. The only new aspect here is that such jobs are now part of a larger compound object. However this does not influence how and what kind of data can be obtained about the individual jobs making up a workflow and thus has no direct influence on the monitoring system.

Generating aggregated information about a workflow as a whole requires understanding about the relation of individual jobs belonging to the workflow. This information is already present in the entity (grid broker, portal or a dedicated workflow service) that supervises the workflow and schedules the jobs, so it is a natural expectation that this component also takes care of receiving information about individual jobs and calculating the required aggregated data. This aggregated data then can be fed back to the monitoring system so others may also make use of it.

Monitoring requirements for supporting workflows thus boil down to providing access to information about individual jobs and allowing the workflow manager component to feed back data to the monitoring system. The monitoring system itself can be completely agnostic about the workflow.

Workflow of services is very similar to the workflow of jobs, it is just based on the capability of monitoring individual services instead of jobs. When using the WS-Resource Framework [13], monitoring of service workflows is even simpler since the workflow itself is represented by a single service that can be monitored.

3.5 Consumer-side requirements

The previous sections described requirements only for the information producers. On the consumer side the most important requirement is knowing what information can be monitored and where. This requires the availability of a registry. The registry may be as simple as a local file or a complex distributed database or P2P service, based on the complexity the monitoring system aims to support. The registry service may be accessed directly by the consumer or there may be an immediate mediator component that extracts only the data relevant for the consumer from the registry.

3.6 Interoperability

There are many monitoring services existing today and unfortunately there is very little (if any) support for interoperability between them. There are many aspects of interoperability including protocols, supported operations, data models and data representation.

There are standardisation forums and bodies like the Global Grid Forum where protocols aimed as standards may be submitted and discussed therefore we deliberately omit protocol issues in this document. Data model and representation issues will be discussed in section 4.2.

4 Proposal for a generic GRID monitoring architecture

In this section we propose a generic grid monitoring architecture. The proposed architecture aims to reach two goals: first, it should be possible to connect existing monitoring systems so they can live together in the same infrastructure. Second, the architecture description aims to help the creators of future monitoring systems.

The description uses the concepts of the original GMA document [3] and extends it with new components and new operations.

4.1 System components

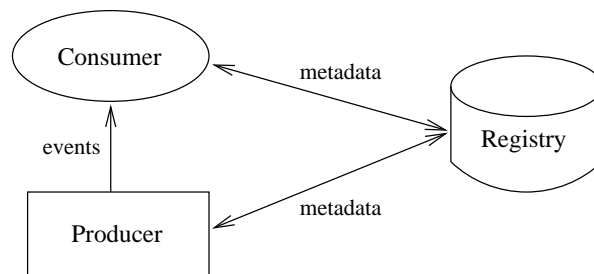


Figure 2: Original GMA components

Our architecture has the basic components defined in the original GMA document (see figure 2) but we both refine and extend them with new components and new operations. These new components and operations are expected to address the shortcomings of the original GMA specification discussed in section 2.1.

The components generating monitoring data are called *sensors*. The data generated by sensors is made available to other grid entities by *producers*. Sensors may be either remote or built into the producer. Producers must have a well-defined interface that may be used by other grid entities while the interface between sensors and producers is implementation-specific and is not exposed to other grid entities.

Actuators are responsible for performing monitoring actions. Just like sensors, actuators are also managed by producers and can be accessed using the interface provided by the producer. It should be noted that the difference between sensors and actuators is more conceptual than technical. Technically, accessing both sensors and actuators means the consumer invoking a remote code fragment on the producer side. The distinguishing difference between sensors and actuators is that measurements performed by sensors are expected not to have a noticeable influence on the monitored entity while actuators are expected to change the state of their target object. Security requirements for sensors and actuators are also different. While it is common that most sensors produce public data accessible by everyone, actuators usually require strict access control. Such access control however is left to the actual implementation and is not defined at this abstract architecture level.

Entities receiving data from producers are called *consumers*. Consumers may also activate actuators. While the term "consumer" might not be the best when talking about interaction with actuators, it is already widely accepted and there is no reason to introduce just another term for actuators.

Although it follows from the above definitions of sensors, producers and consumers, it is worth mentioning that a given software component may function both as a consumer and a producer at the same time. This allows the deployment of complex multi-level data routing and distributed processing setups. Also, a component may be a consumer of one

monitoring system and a producer for some other monitoring system thus acting as a gateway between different monitoring system implementations.

The *registry* is an index service or directory service that holds information about available producers and consumers. It holds the metadata that is necessary to locate a producer or consumer and to find out its capabilities. There are no requirements about the implementation of the registry. It may be anything from a local file to a distributed database or P2P network depending on the complexity of the monitoring system implementation.

The *mediator* is a new component compared to the GMA [3]. It handles consumer and producer registration and updates the metadata in the registry accordingly. The mediator actively tries to match consumer and producer registrations and notifies both parties when a match is found. The mediator however does not influence the direct communication between a producer and a consumer.

The mediator may also have brokering features, like handling the complete end-to-end path for monitoring data, or creating new components needed for data path establishment (like a gateway service if the consumer and producer belong to different monitoring systems).

Searching for a match between producer and consumer registrations may be both time and resource consuming and requires non-trivial understanding of the registered metadata therefore it may be desirable to implement the mediator as a stand-alone service. We do not require this at the architecture level however to allow simple mediator implementations built directly into consumers or producers.

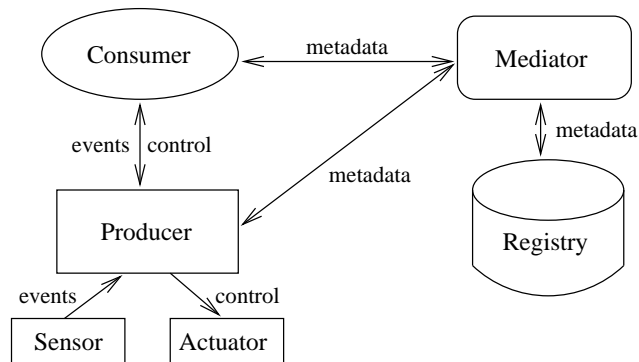


Figure 3: C-GMA components

Figure 3 shows the C-GMA components and their interactions.

4.2 Data model

To ensure maximum interoperability the architecture specification does not enforce any specific data model. However there are some requirements that must be met in order to allow interoperability between different monitoring systems.

Following the GMA [3] model, producers send monitoring data to consumers in the form of time-based *events*. Every event must have a well-defined schema. The schema can be either static or highly dynamic (for example, applications may register their own schema components that are only valid while the application is running).

The monitoring system must provide an interface for consumers to discover the schema being used by a producer. If the producer supports a static schema only it is sufficient to publish the schema to a registry so the producers do not need to provide a schema management interface. With a dynamic schema it may not be desirable to publish schema changes if they happen very frequently; in this case, the consumer must be able to query the current schema for the data it requests directly from the producer.

There is no generic requirement about the data representation. Using self-expressing formats like XML makes interoperability easier but has considerable overhead both in the amount of data to transfer and the resource usage during processing. Using a compact, proprietary format eliminates the overhead but makes interoperability harder. A monitoring system may support multiple different data representations in order to adapt to both requirements.

4.3 Component interactions

The model of communication between producers and consumers follows the original GMA [3] proposal, with the addition of a new control/response interaction.

The publish/subscribe and query/response interactions described in the original GMA are restricted to interact with sensors only. The control/response interaction is similar to the query/response interaction but it invokes an actuator instead of a sensor. Just like query/response, control/response must be initiated by a consumer. The first stage of the interaction sets up the transfer and invokes the actuator on the producer side. The producer then returns the result of the actuator invocation (usually a status code or status message) in a single response. The implementation may allow the definition of actuators that do not send back any result to the consumer.

Consumers and the producers now interact with the mediator instead of the registry during the registration and discovery of the corresponding component. A monitoring session then can be broken up to four phases (naming is adopted from Condor Matchmaking [16]):

Advertising – registration of producers and consumers. Registration is soft-state, components must renew registration before expiration.

Matching – based on registered metadata, the mediator looks for matching pairs. When a new pair is found, both parties are informed about corresponding one.

Claiming – direct communication between producer and consumer (occurs when a component is notified about a corresponding component). Mutual compatibility between components, verification of capabilities and data requirements must be done in this phase.

It is expected that during claiming phase, native protocol between producer and consumer will be used. However, the C-GMA specification also contains definitions of optional claiming operations. In some cases (i. e. for lightweight implementations), it may be beneficial to delegate claiming to the mediator.

Data transfer – data (events) are sent directly between producer and consumer in their native protocol.

4.3.1 Defined operations

Compared to the GMA, the list of functions supported by the producer is extended with new operations to support the control/response interaction and communication with the mediator.

Accept Control – accept an actuator invocation request from a consumer. After invoking the actuator, zero or more events are sent to the consumer in response. Corresponds to Consumer *Invoke Control*.

Accept MatchNotify – accept a notification message from the mediator. In response the producer may enter the claiming phase to initiate communication with the consumer.

QueryCap – query the consumer for its list of capabilities. Optional.

Accept QueryCap – accept capability request during the claiming phase. The list of capabilities supported by the producer is sent back as part of the reply. Optional.

Similarly the consumer interface is extended with

Invoke Control – request the invocation of an actuator from the producer. Any response from the actuator is delivered as an event as part of the reply. Corresponds to Producer *Accept Control*.

Accept MatchNotify – accept a notification message from the mediator. In response the consumer may enter the claiming phase to initiate communication with the producer.

QueryCap – query the producer for its list of capabilities. Optional.

Accept QueryCap – accept capability request during the claiming phase. The list of capabilities supported by the consumer is sent back as part of the reply. Optional.

Core functions supported by the mediator are listed below:

- Advertising part (mandatory):

Add – accept registration from a producer or a consumer. The mediator is responsible for registry maintenance. The matchmaking process can be started based on the new registration.

Update – update a registration. Since registration is soft-state, components must periodically update their registration, otherwise the registration is discarded.

Remove – accept removal request from a producer or a consumer. The registration is discarded immediately.

- Notification part (mandatory):

MatchNotify – when a new matching consumer/producer pair is found, the mediator sends notification about the corresponding peer to both parties. It is up to the notified component whether claiming will be started or not. Notifications are sent asynchronously and have incremental semantics (i.e. a new notification does not invalidate previous ones). Corresponds to **Accept MatchNotify** on the producer and consumer side.

Search – accept a query from a producer or a consumer. All matching components are returned. Expected usage of this operation is during component (re)start. Corresponds to **Locate Consumer** on the producer and **Locate Producer** on the consumer side.

- Claiming part (optional, only if claiming is supported by the mediator itself):

QueryCap – query the remote component for capabilities. The list of the peer's capabilities is returned. Used during the claiming phase.

See Appendix A for a complete list of operations supported by the C-GMA components.

4.4 Capability and attribute management

The generic monitoring architecture must accommodate the co-existence of different components which are either completely incompatible or specialised for different purposes. This requirement is addressed by introducing another metadata layer - component capabilities and data attributes.

In order to prevent confusion we emphasise that neither data attributes nor component capabilities are related to the event data types. On the contrary, data schema is managed according to GMA, at the C-GMA data layer. Hence capabilities and attributes are properties orthogonal to data types.

4.4.1 Component capabilities

Every component declares via their *capabilities* any features that may affect either the possibility of communication with other components or their ability to handle particular data. Component capabilities may include

- the protocol(s) the component speaks,
- authentication and security mechanisms the component supports,
- level of persistence – once a data event is accepted, it is guaranteed not to be lost e. g. in case of a machine crash,
- level of trustworthiness – which class of sensitive data can be sent to the component.

4.4.2 Data attributes

Attributes represent the meta-description of data expressing in which way and to which components the concrete data may be handed over. Data attributes may include

- preciousness – this data may not be lost, i.e. it should be handled by "persistent" components only
- sensitiveness – expresses the level of trustworthiness required from components to handle the data

4.4.3 Capability language

Both capabilities and attributes are expressed using a *capability language*. Having a common language for both of these entities allows treating them in a symmetric way, namely expressing requirements on capabilities using attributes and vice versa. At the architecture level we do not prescribe a specific language. Section 4.5 proposes two possible capability languages.

4.4.4 Attribute scope

Unlike capabilities, which are always assigned to a single component instance, attributes may occur at three levels (from the most general to the most specific):

- data type – bound to a specific data type; any data of this type carries these attributes,
- component – bound to a specific component (producer); any data produced by this component carries these attributes,
- data record – any piece of data (monitoring event) may carry its specific attributes.

However, none of these levels is mandatory – a C-GMA implementation may choose to support only some of them¹. On the other hand, when an implementation allows more than one level of attribute specification, conflicts may arise. Hence in this case the capability language must provide the operation of *attribute override* – given two attribute specifications (one more general, e. g. component level, the other more specific, e. g. data record level) it yields a single consistent attribute specification at the more specific level.

¹But supporting at least component-level attributes is encouraged to benefit from all C-GMA features

4.4.5 Attribute and capability semantics

Capabilities and attributes are processed by mediator components in order to match producers with consumers. However, the architecture does not define any concrete capabilities and attributes nor does it require any specific interpretation. For C-GMA these are semantics-free objects which are manipulated in a pure syntactic way (e. g. compared for equality). Concrete meaning of capabilities/attributes is known and evaluated only by concrete producers and consumers.

4.4.6 Security issues

When evaluating attributes/capabilities by producers and consumers, certain level of trust may be required. Namely the following issues have to be addressed:

- Two components that are going to communicate with each other have to make sure that they both interpret a given attribute/capability in the same way.
- When a component advertises a capability, the other party has to be able to verify that the capability is not forged, e. g. that sensitive data will not be passed to a non-trustworthy component.

Both these issues can be addressed in a given infrastructure by attaching some sort of digital signatures to advertised capabilities and attributes, together with a clear definition of attribute/capability namespaces and assigning trusted certification authorities to them. However, for the sake of generality, we do not require any specific security mechanism; we even don't require any such mechanism to be present.

4.5 Layered system architecture

In the previous section we described the C-GMA concepts at an abstract level. However, our goal is providing an infrastructure which ensures interoperability of components, even if those come from different independent original implementations. For achieving interoperability a strict definition of common interfaces is required, though.

On the other hand, we still want to impose minimal requirements on the involved components, in particular we do not want to put any restrictions on the data description, representation and queries the components use.

Therefore the full specification of a C-GMA compliant monitoring infrastructure is two-layered:

1. The *capability layer* defines the language used to express component capabilities and data attributes, as well as concrete operations of all infrastructure components (i. e. mediator, producer and consumer).
2. The *data layer* defines the language of describing, expressing and querying data.

In this section we describe requirements on these two layers of the C-GMA specification. Section 5 shows two concrete proposals of the capability layer definition (i. e. two independent, non-interoperable C-GMA worlds), while section 6 deals with examples of data-layer definitions.

4.5.1 Capability layer

The capability language is unique and common for all the components that are supposed to be part of the cooperating infrastructure. There are two mandatory operations the language must support:

Component matching – given the capabilities of a producer and a consumer it must be possible to decide whether these components can communicate with each other, e. g. whether they implement the same protocol.

Attribute matching – given attributes of a piece of data and capabilities of both the producer and the consumer it must be possible to decide whether this producer may handle this data over to this consumer, e. g. whether data security requirements are satisfied.

In addition, if the infrastructure supports attribute specification at multiple levels (see section 4.4.4), the following operation must be defined:

Attribute override – given two attribute specifications, one more general, the other more specific, merge them together and resolve possible conflicts. E. g. when one attribute is defined at multiple levels, it must be clear whether one of them takes precedence or all should be used to form a multivalued attribute etc.

As semantics of attributes is not known at this level, the capability language must contain e. g. directives to determine override behaviour for specific attributes.

Once the capability language is fixed it is possible to define exactly the operations of components, namely the mediator. The operations can be defined either in terms of communication protocol, e. g. WSDL's of the components, or as the operation binding(s) in concrete programming languages.

In order to be complete the operation definitions must include arguments referring to items of data language. However, the data language is not only undefined at this layer yet, but it is also desirable not to bind the operations to a concrete data language (so that multiple "data worlds" may coexist within a single infrastructure, sharing the common capability language). Hence we treat the data-related arguments here as opaque, either string or binary objects. Their interpretation is left to concrete producers/consumers and the data-specific modules of mediator respectively.

4.5.2 Data layer

The purpose of the data-layer language is providing means of describing and accessing data. A full definition of data language must be able to specify:

Data types identify uniquely what is the data entity a producer generates or a consumer requests.

E. g., in the case of relational data model, it is a name of the table as well as names and types of individual columns.

Actual data transferred between producers and consumers.

In addition, in many real applications it is likely to be desirable to include also

Conditions express a restriction on the domain given by a concrete data type. The producer declares with these conditions that it produces only data satisfying the conditions, the consumer expresses its interest only in such data and not others.

In the relational data model the conditions are WHERE clauses of the producer and consumer defining SQL statements.

4.5.3 Notes on interoperability

The C-GMA design assumes that there are multiple data definitions of various data languages all taking part in a single logical infrastructure, sharing the common capability layer definition.

Our goal is that these, otherwise independent, "data worlds" co-exist within a single infrastructure given by the common capability-layer definition. In this sense we consider two components of different data worlds to be interoperable if they are both able to register within a single infrastructure.

Quite obviously, this level of interoperability does not imply directly that those components are able to exchange data, i. e. achieve the interoperability in the more conventional sense.

However, the existence of common infrastructure allows the following:

- Through their capabilities, the components are able to advertise to which data world they belong. Hence they may all benefit from the services provided by the single infrastructure (registration and matching) while being independent otherwise.
- There may exist joint "bridge" components having interfaces to more data worlds. Through such bridges data may flow from one data world to another, achieving the conventional interoperability, then.

5 Proposed capability languages

5.1 Condor ClassAds

For description of service capabilities and data attributes, ClassAd language [16] is used. Each of producer capabilities, consumer capabilities and data attributes will be represented by one ClassAd. ClassAds may contain static attributes and capability values and also explicit requirements on matching component.

For matching, standard matching is used. To allow matching of three ClassAds, composed ClassAd which consists from ClassAds of producer, consumer and data attributes is used in form:

```
{
    Producer = {
        ...
    }
    Data = {
        ...
    }
    Consumer = {
        ...
    }
}
```

Requirements from all three sub-ClassAds are evaluated in context of this composed ClassAd.

5.1.1 Simple example

```
Producer = {
    Protocol = { http, https};
}
```

```

Data = {
    MinSecLevel = 4;
    Requirements = (.Consumer.SecurityLevel >= MinSecLevel);
}

Consumer = {
    Protocol = {https};
    SecurityLevel = 5;
    Requirements = member(Protocol, .Producer.Protocol);
}

```

The example illustrates the description of static capabilities and attributes and explicit requirements of consumer and data. References to attributes/capabilities of compared ClassAd are also demonstrated.

5.2 XML XPath based capability language

The principal idea of this proposed capability language is describing both data attributes and component capabilities in terms of XML documents. Consequently XPath [21] expressions can be used to localise items in these documents as well as express various conditions.

The attribute/capability XML documents contain:

- constant attribute/capability values
- requirements on the other parties
- ranking expressions in the future (not being discussed here yet)

We considered two different approaches to the document schema: flat and structured constant values. Probably the structured approach is more general. However, the flat one is considerably easier to be handled in implementation while not restricting any of the principal C-GMA features. Therefore we stick with the flat model for the time being.

For the sake of simplicity we also assume that data attributes are specified at the component-level only. Consequently no override operation is required (see section 4.4.4).

5.2.1 Constant attribute and capability values

Component capabilities are expressed as XML documents having either `<producer>` or `<consumer>` as the root element, e. g.

```

<producer>
  <cap name="protocol">A</cap>
  <cap name="encoding">AES</cap>
  <cap name="encoding">DES</cap>
  ...
</producer>

```

where the `<cap>` element with the mandatory attribute name records constant capability values. Multi-valued capabilities are stored as multiple occurrences of the element.

On the other hand, data attributes are expressed in a very similar form, a document with `<data>` root element and multiple `<attr>` elements inside.

The content of `<cap>` and `<attr>` elements must be CDATA in the considered flat model. There are also optional attributes `common-cap` and `common-attr` to specify simplified requirements, see section 5.2.3.

5.2.2 Explicit requirements

Relationship between capabilities and attributes is expressed in terms of `<req>` elements. For the purpose of matching the individual producer, consumer and data capability/attribute documents are assumed to be merged together using the following XInclude [22] root document:

```
<match xmlns:xi="http://www.w3.org/2003/XInclude">
  <xi:include href="producer.xml" parse="xml"/>
  <xi:include href="consumer.xml" parse="xml"/>
  <xi:include href="data.xml" parse="xml"/>
</match>
```

The `<req>` element contains a mandatory attribute test which is an XPath expression being evaluated with the root `<match>` element of the above document as the starting context node.

The three documents `producer.xml`, `consumer.xml`, and `data.xml` match mutually if and only if all the `<req>` expressions are evaluated to true.

5.2.3 Implicit requirements

In addition to expressing requirements via the `<req>` elements described above we define shortcuts (i. e. they can be translated to `<req>`'s in a straightforward way) for testing attribute/capability values for equality.

Either `<cap>` or `<attr>` elements may specify additional attribute `common-cap` and `common-attr` having the value of "producer", "consumer", or "data". In this way existence of `<cap>` or `<attr>` in the referred-to party is required, having the same value as the referring element.

Currently, the value of `common-cap` and `common-attr` is redundant - from the context it is always clear which party is referred to, i. e. it might have boolean values only. However, it is defined for future extensions.

5.2.4 Simple example

```
<producer>
  <cap name="protocol" common-cap="consumer">A</cap>
</producer>

<consumer>
  <cap name="protocol" common-cap="producer">A</cap>
  <cap name="securityLevel">5</cap>
</consumer>

<data>
  <!-- without "common", i.e. private -->
  <attr name="minSecLevel">4</attr>

  <req test="consumer/cap[@name='securityLevel'] >=
    data/attr[@name='minSecLevel']" />
</data>
```

The example illustrates both explicit and implicit requirements. Both producer and consumer require the other component to speak the same protocol via implicit requirement (`common-cap` attribute). In addition, data require the consumer to satisfy certain security condition via an explicit `<req>` expression.

5.2.5 Mapping of core functions

As explained in section 5 an exact specification of component interfaces is done at the capability layer in order to allow the interoperability.

For an infrastructure based on the XML-based capability language defined in this section we assume, on the contrary to the ClassAd infrastructure (section 5.1), the components implementation to be hidden behind a well-defined API. The C-GMA core functions are mapped to API calls, usually pairwise, i. e. a pair of corresponding core functions is implemented with a client-side API call, while the potential communication with the server is hidden in the API implementation, and server-side callback function eventually.

Despite this approach may look less flexible (e. g. a restricted programming language binding), it allows more implementation flexibility, e. g. the mediator can be seamlessly implemented as a distributed service.

The API exports the following C-GMA-specific calls to the mediator service:

cgma_registration implements MaintainRegistration at the producer/consumer side and Add, Update, and Remove at the mediator side.

cgma_search implements LocateConsumer (LocateProducer) at the consumer (producer) side, and Search on mediator.

On the other hand, the following calls are used to activate producer and consumer functions:

cgma_match_notify implements MatchNotify on the mediator and AcceptMatchNotify on the producer/consumer. It is called by the mediator for each pair of matching components found. It is the responsibility of the API implementation to deliver appropriate messages to both the involved components.

cgma_query_cap implements the pair QueryCap/AcceptQueryCap symmetrically for a producer to query consumer's capabilities or vice versa.

The rest of producer and consumer core functions is related to their data communication; therefore it is left to be specified in data-layer protocols.

6 Considered data models

6.1 Trivial relational language

The data language presented in this section is very simple and artificial by intention. It is defined for proof-of-concept only, we want to demonstrate the important C-GMA features avoiding implementation problems of complex real data languages.

The language is relational, i. e. the principal data entity is a table with a fixed number of named and typed columns. Rows of the tables are GMA data events passed from producer to consumer (cf. R-GMA [7]). Querying capabilities are limited, there is no equivalent of join operation, and filtering conditions are restricted too.

6.1.1 Data types

For a particular table **XXX** the following XML Schema types are defined:

```
<xs:complexType name="XXXRow">
  <xs:sequence>
    <xs:element name="YYY" type="xs:TYPE"
      nillable="true/false"
      minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

```

    ...
  </xs:sequence>
</xs:complexType>

<xs:complexType name="XXX">
  <xs:sequence>
    <xs:element name="row" type="XXXRow"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

```

where TYPE is further restricted to int, string, or dateTime. An element of the resulting type XXXRow represents a single row of the table XXX (i. e. GMA event) while the type XXX represents a fragment of the table.

6.1.2 Filters

Producers describe the table fragments they produce as well as consumers declare the data they are interested in in terms of filters. A filter can basically express a very restricted subset of SQL SELECT statements:

```

<filter table="XXX">
  <conditions column="YYY">
    <compare op="OP1">VALUE1</compare>
    <compare op="OP2">VALUE2</compare>
    ...
  </conditions>
  <conditions column="ZZZ">
    <compare op="OP3">VALUE3</compare>
    <compare op="OP4">VALUE4</compare>
    ...
  </conditions>
  ...
</filter>

```

where OP's are one of \neq , $<$, $=$, $>$. The shown filter has the same semantics as the following SQL statement:

```

select * from XXX
where (YYY OP1 VALUE1 or YYY OP2 VALUE2)
and (ZZZ OP3 VALUE3 or ZZZ OP4 VALUE4)

```

i. e. always all columns of a table are selected, we allow only two-level nesting of logical expression, the conditions on the bottom level must refer to the same column and are logically or-ed, the conditions on the upper level are logically and-ed.

6.1.3 Component interaction

Both producers and consumers advertise with a table name (besides attributes and capabilities) in the simplest case, or with a filter (which may be simplified, i. e. covering a larger result set). We assume the schema is well-known, i. e. all components can map a table name to the list of its columns and their types.

Data part of matching is done by looking up the potential party based on table name first. In the more complicated case when filters are advertised it is refined further by checking the union of filters (i. e. anding them) for logical contradiction. Due to the restricted structure of the filter formula the checking is trivial.

On claiming the producer and consumer must check the table name, and optionally evaluate the filters (which are fully specified now, unlike the matching phase when empty or simplified filters may be involved) for logical contradiction to avoid keeping unnecessary connection.

Data transfers send XML documents of the type XXX (see section 6.1.1).

6.2 Interface to the Mercury Monitoring System

6.2.1 Data types

The Mercury Monitoring System [5] uses a simple language for defining the type of monitored data. The data model used by Mercury is based on *metrics*. Every sensor in Mercury can provide one or more metrics. Every metric has a unique name, a well-defined data type and a set of parameters. Every monitoring event generated by the producer belongs to a single metric.

Mercury has no distributed registry for the data schema therefore connected producer and consumer must agree on the definition of the schema. On the other hand consumers can query the schema from the producer run-time therefore dynamic schema changes are possible. Indeed, it is even possible for a consumer to simultaneously talk to several producers having different schemes.

Mercury installations are usually multi-level: every monitored machine runs a so-called Local Monitor while the cluster's frontend runs a Main Monitor. The Main Monitor accepts requests from consumers outside of the cluster and dispatches these requests to the individual Local Monitors and then routes the responses back to the consumer. The Main Monitor (or more specifically, the routing sensor inside the Main Monitor) uses the metric parameters to decide which Local Monitor is the request targeted at.

This represents a problem with metric parameters: we either

- advertise just the existence of parameters therefore leaving it up to the consumers to interpret them, or
- advertise all possible values (which may not be possible if the range of allowed values is itself dynamic).

Both kind of advertisements may be present in the same registry simultaneously thus facilitating the requirements of both simple and advanced consumers. For simplicity we demonstrate here only the second option. Therefore representing a Mercury data source in C-GMA we need a tuple with the following elements:

- the location (URL) of the producer where the metric is available
- name of the metric
- value of metric parameters (host name for host-based metrics, grid job identifier for application-specific metrics)
- data type described as a string

Mercury also supports actuators. Just like metrics, the single operations that can be requested from an actuator are defined by *controls*. The definition of a control has the same format as metric definitions and therefore can be represented in C-GMA the same way.

6.2.2 Queries, searching

Following the data definition the query language is again a tuple. For simplicity we only include the metric name, parameter values and data type allowing POSIX 1003.2 regular expression matching for filtering. Indeed, the producer location can be considered technical detail of the data layer, needed for building the communication channel but not strictly necessary for defining the query's semantics.

6.2.3 Component interaction

Producers *advertise* the tuples with producer location, metric name, parameter values and data type. Consumers *advertise* query tuples. Since the schema is not defined globally and can also change dynamically, data types are handled as strings on the C-GMA level. For *claiming* and *data transfer* the native Mercury protocol is used between the consumer and the producer.

Allowing the consumer to use regular expressions in the type component of the query tuple makes it possible to adapt for dynamic schema changes. On the other hand, using a specific type in the query registration provides schema compatibility checking in the capability layer.

6.2.4 Example

Examples about data sources registered by producers:

```
(skirit.ics.muni.cz; pbs.queueelen; queue=skirit; int)
(skirit.ics.muni.cz; host.loadavg;
  host=skirit1.ics.muni.cz; double[3])
(skirit.ics.muni.cz; host.loadavg;
  host=skirit2.ics.muni.cz; double[3])
(n0.hpcc.sztaki.hu; app.cputime; jobid=gridjob001; double)
```

The above example shows the registration of two producers (`skirit.ics.muni.cz` and `n0.hpcc.sztaki.hu`). Information about individual hosts in the `skirit` cluster can be requested from `skirit.ics.muni.cz`. The consumers should not be aware how the requests are routed to the individual hosts. Similarly, `n0.hpcc.sztaki.hu` offers the monitoring of the CPU time consumed by the job identified by `gridjob001`; the consumer does not need to know on which internal nodes the job is really running.

The following example shows consumer-registered queries:

```
(host.loadavg; host=skirit.*.ics.muni.cz, double[3])
(app.cputime; jobid=gridjob001, .*)
```

The first example shows a consumer that is interested in load average information for all hosts in the `skirit` cluster. Since the expected data type is fully specified, a producer exporting the same data but in a different format will not match the query. On the contrary, the consumer emitting the second query is only interested in the name of the metric and is expected to deal with whatever data type the application actually registered for it.

7 Plans for the prototype implementation and future research

The expected prototype implementation will consist of the following elements:

- a basic mediator service with web-service interface
- a central registry, possibly included in the mediator service itself
- ClassAd-based capability language
- Mercury and a simple SQL-based language as data layers

Independently, peer-to-peer implementation of mediator and registry is studied in cooperation with INFN. Foreseen capability language is XML based, with data languages reused from the first prototype implementation.

Based on this proof-of-concept implementations, possible revision of defined operations may appear. Research may continue in several areas:

- implementation of C-GMA using complete WSRF and WS-Notification standards,
- proposing an API for the monitoring system components,
- distributed implementation of the registry,
- relationship between peer-to-peer networks and C-GMA. Mediator can be implemented as library which does mediator tasks using peer-to-peer implementation of registry,
- advance features of mediator, including maintenance of end-to-end data channels and possibilities to create new components needed to establish such channel.
- constraint-based capability language,
- using ideas from semantic web for advance matching of resources [23],
- security - verification of capabilities, privilege delegation.

8 Conclusion

In this document we described a generic monitoring architecture suited for integrating existing monitoring solutions in the same infrastructure. We also discussed problems and requirements for monitoring system implementations. We proposed two possible realisations of both architecture layers: a ClassAd based and an XML/XPath based capability language for the capability layer, and Mercury and a simple SQL-based language for the data layer.

We believe that the architecture outlined in this document is suitable for alleviating interoperability problems between existing monitoring solutions to the extent that multiple monitoring systems can live within the same infrastructure. However several interoperability problems like standardised protocols remain unsolved therefore future research in this area is still needed.

References

- [1] Michael Gerndt, Roland Wismüller, Zoltán Balaton, Gábor Gombás, Péter Kacsuk, Zsolt Németh, Norbert Podhorszki, Hong-Linh Truong, Thomas Fahringer, Marian Bubak, Erwin Laure, and Thomas Margalef. Performance tools for the grid: State of the art and future. Technical report, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation – Technische Universitaet Muenchen (LRR-TUM), 2004.
- [2] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163–188, Jan 2005.
- [3] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture. Technical Report GWD-PERF-16-2, Global Grid Forum, Jan 2002.
- [4] Zoltán Balaton, Péter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. From cluster monitoring to grid monitoring based on grm. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001*, volume 2150 of *Lecture Notes in Computer Science*, pages 874–881. Springer, 2001.
- [5] Zoltán Balaton and Gábor Gombás. Resource and job monitoring in the grid. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 404–411, Klagenfurt, Austria, August 26–29 2003. Springer-Verlag.
- [6] The GridLab Project. <http://www.gridlab.org>.
- [7] Steve Fisher. Relational model for information and monitoring. Technical Report GWD-PERF-7-1, Global Grid Forum, 2001.
- [8] Mark A. Baker and Matthew Grove. jGMA: A lightweight implementation of the Grid Monitoring Architecture. Technical report, Distributed Systems Group, University of Portsmouth, Sep 2004.
- [9] Mark A. Baker and Matthew Grove. jGMA: A lightweight implementation of the grid monitoring architecture. In *Proceedings of UKUUG LISA/Winter Conference 2004*, Bournemouth, UK, Feb 2004.
- [10] pyGMA. <http://www.didc.lbl.gov/pyGMA>.
- [11] Warren Smith. A system for monitoring and management of computational grids. In *ICPP 2002*, pages 55–64, Vancouver, Canada, August 18–21 2002. IEEE Computer Society.
- [12] Ws-notification. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.
- [13] Ws-resource framework. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.
- [14] Global grid forum. <http://www.ggf.org>.
- [15] Ian Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J.Treadwell, and J. von Reich. The open grid services architecture, version 1.0. Informational Document GFD-I.030, Global Grid Forum, Jan 2005.
- [16] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC'98*, pages 140–147, Chicago, IL, July 28–31 1998. IEEE Computer Society.

- [17] Chuang Liu, Lingyun Yang, Ian Foster, and Dave Angulo. Design and evaluation of a resource selection framework for grid applications. In *HPDC-11*, pages 63–72, Edinburgh, Scotland, UK, 2002. IEEE Computer Society.
- [18] Rajesh Raman. Matchmaking frameworks for distributed resource management. Technical report, Computer Science, University of Wisconsin, Madison, 2000.
- [19] Chuang Liu and Ian Foster. A constraint language approach to grid resource selection. http://www.cs.uchicago.edu/files/tr_authentic/TR-2003-07.pdf.
- [20] Bartosz Baliś, Marian Bubak, Marcin Radecki, Tomasz Szepieniec, and Roland Wismüller. Application monitoring in crossgrid and other grid projects. In Marios D. Dikaiakos, editor, *Grid Computing – Second European AcrossGrids Conference*, volume 3165 of *Lecture Notes in Computer Science*, Nicosia, Cyprus, January 28–30 2004. Springer–Verlag.
- [21] Xpath specification. <http://www.w3.org/TR/xpath/>.
- [22] Xinclude specification. <http://www.w3.org/TR/xinclude/>.
- [23] Hongsuda Tangmunarunkit, Stefan Decker, and Carl Kesselman. Ontology-based resource matching in the grid - the grid meets the semantic web. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 706–721. Springer, 2003.

A Complete list of operations for C-GMA components

A.1 Functions supported by the producer

A.1.1 Data transfer operations

Initiate Subscribe – request the consumer to accept events.

Initiate Unsubscribe – terminate a subscription at a consumer.

Accept Subscribe – accept subscription request from a consumer.

Accept Unsubscribe – accept subscription termination request from a consumer.

Accept Query – accept a query from the consumer.

Accept Control – accept an actuator invocation request from a consumer. After invoking the actuator, zero or more events are sent to the consumer in response. Corresponds to Consumer *Invoke Control*.

Notify – send a single set of events to a consumer.

A.1.2 Registration and claiming protocol

Maintain Registration – add, update or remove registration. Corresponds with Add, Update and Remove operations on the mediator.

Locate Consumer – query the mediator for all consumers matching the producer's requirements. Corresponds with Search on the mediator.

Accept MatchNotify – accept a notification message from the mediator. In response the producer may enter the claiming phase to initiate communication with the consumer.

QueryCap – query the consumer for its list of capabilities. Optional.

Accept QueryCap – accept capability request during the claiming phase. The list of capabilities supported by the producer is sent back as part of the reply. Optional.

A.2 Functions supported by the consumer

A.2.1 Data transfer operations

Initiate Subscribe – request the producer to send events.

Initiate Unsubscribe – terminate a subscription at the producer.

Accept Subscribe – accept subscription request from a producer.

Accept Unsubscribe – accept subscription termination request from a producer.

Initiate Query – request events from the producer. Corresponds with Accept Query.

Invoke Control – request the invocation of an actuator from the producer. Any response from the actuator is delivered as an event as part of the reply. Corresponds to Producer *Accept Control*.

Accept Notify – accept a single set of events from the producer.

A.2.2 Registration and claiming protocol

Maintain Registration – add, update or remove registration. Corresponds with Add, Update and Remove operations on the mediator.

Locate Producer – query the mediator for all producers matching the consumer's requirements. Corresponds with Search on the mediator.

Accept MatchNotify – accept a notification message from the mediator. In response the consumer may enter the claiming phase to initiate communication with the producer.

QueryCap – query the producer for its list of capabilities. Optional.

Accept QueryCap – accept capability request during the claiming phase. The list of capabilities supported by the consumer is sent back as part of the reply. Optional.

A.3 Functions supported by the registry

Add – register a component in the directory.

Update – update the registration of a component.

Remove – remove a component from the directory.

Search – perform a search for a component based on some search criteria.

A.4 Functions supported by the mediator

A.4.1 Advertising

These operations are mandatory.

Add – accept registration from a producer or a consumer. The mediator is responsible for registry maintenance. The matchmaking process can be started based on the new registration.

Update – update a registration. Since registration is soft-state, components must periodically update their registration, otherwise the registration is discarded.

Remove – accept removal request from a producer or a consumer. The registration is discarded immediately.

A.4.2 Notification

These operations are mandatory.

MatchNotify – when a new matching consumer/producer pair is found, the mediator sends notification about the corresponding peer to both parties. It is up to the notified component whether claiming will be started or not. Notifications are sent asynchronously and have incremental semantics (i.e. a new notification does not invalidate previous ones). Corresponds to **Accept MatchNotify** on the producer and consumer side.

Search – accept a query from a producer or a consumer. All matching components are returned. Expected usage of this operation is during component (re)start. Corresponds to **Locate Consumer** on the producer and **Locate Producer** on the consumer side.

A.4.3 Claiming phase

These operations are optional.

QueryCap – query the remote component for capabilities. The list of the peer’s capabilities is returned. Used during the claiming phase.