A Web Service-based Experiment Management System for the Grids*

Radu Prodan and Thomas Fahringer
Institute for Software Science
University of Vienna
Liechtensteinstrasse 22
A-1090 Vienna, Austria
{radu,tf}@par.univie.ac.at

ABSTRACT

We have developed ZENTURIO, which is an experiment management system for performance and parameter studies as well as software testing for cluster and Grid architectures. In this paper we describe our experience with developing ZENTURIO as a collection of Web services. A directive-based language called ZEN is used to annotate arbitrary files and specify arbitrary application parameters. An Experiment Generator Web service parses annotated application files and generates appropriate codes for experiments. An Experiment Executor Web service compiles, executes, and monitors experiments on a single or a set of local machines on the Grid. Factory and Registry services are employed to create and register Web services, respectively. An event infrastructure has been customised to support high-level events under ZENTURIO in order to avoid expensive polling and to detect important system and application status information. A graphical user portal allows the user to generate, control, and monitor experiments. We compare our design with the Open Grid Service Architecture (OGSA) and highlight similarities and differences. We report results of using ZENTURIO to conduct performance analysis of a material science code that executes on the Grid under the Globus Grid infrastructure.

1. INTRODUCTION

Computational Grids have become an important asset aimed at enabling programmers and application developers to aggregate resources scattered around the globe for large-scale scientific and engineering research. However, developing applications that can effectively utilise the Grid still remains very difficult because of the lack of sophisticated tools to support developers.

Over the years, distributed object-oriented technologies, such as the Java Remote Method Invocation (RMI [16]), the Common Object Request Broker Architecture (CORBA [19]), the Distributed Component Object Model (DCOM [5]), or Jini [10], have emerged as popular distributed computing standards, which meet the requirements of building Grid services. In the year 2000, a consortium of companies comprising Microsoft, IBM, BEA Systems, and Intel defined a new set of XML [17] standards for programming Business-to-Business (B2B) applications called Web services, which are being standardised under the umbrella of the W3C consortium [32]. Web services address heterogeneous distributed computing by defining techniques for describing software components, methods for accessing these components, and discovery methods that enable the identification of relevant service providers. A key advantage of Web services is their programming language, model, network, and system software neutrality.

In previous work [21] we introduced the functionality of ZENTURIO, which is an experiment management system for cluster and Grid computing that alleviates parameter studies, performance analysis, and software testing. ZENTURIO uses a directive-based language called ZEN [20] to specify arbitrarily complex program executions. ZEN enables the programmer to invoke experiments for arbitrary value ranges of any problem parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. Moreover, ZEN directives can be used to request for a large variety of performance metrics (e.g. cache misses, load imballance, execution, communication, synchronisation time).

This paper describes experiences with using a Web application and services toolkit to implement and deploy ZENTURIO as a Grid tool, consisting of a collection of distributed Web services. An Experiment Generator Web Service parses application files annotated with ZEN directives and generates appropriate application codes based on the semantics of the directives encountered. An Experiment Executor Web Service retrieves a set of experiments and compiles, executes, and monitors them on the target machine. Upon completion of experiments, the output files and performance data are stored into a data repository for post-mortem multi-experiment performance and parameter studies. Factory and a Registry Web Services are employed to create and register Web services, respectively. An asynchronous event framework enables ZENTURIO Web Services to notify clients about interesting system and application events. This is important to avoid expensive polling and to detect important status information about the system and application. A graphical User Portal enables the user to create, control, and monitor experiments as they progress. Various graphical diagrams are provided to visualise output and performance data.

The paper is organised as follows: Section 2 presents a brief overview of the ZENTURIO functionality. Section 3 describes the Web service-based architecture in detail. The next section is devoted to the events supported by ZENTU-

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

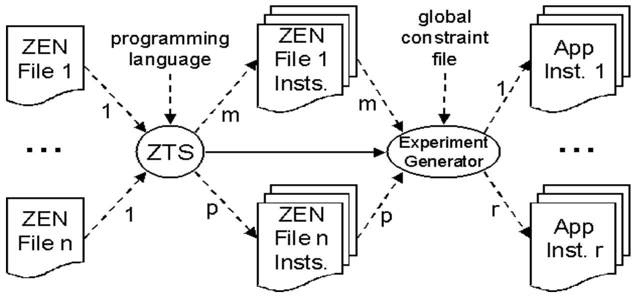


Figure 1: The ZEN application instance generation.

RIO. In Section 5 we compare our design with the Open Grid Service Architecture (OGSA) [14, 28]. A performance study of a material science code that has been conducted by ZENTURIO and executed under Globus [13] is presented in Section 6. Section 7 concludes the paper.

2. ZENTURIO OVERVIEW

ZENTURIO is a system which enables the user to automatically conduct a large number of experiments (up to thousands) on cluster and Grid architectures. Its goal is to support performance analysis and tuning, parameter studies, and software testing.

Conventional parameter study tools [1, 34] restrict parameterisation to input files only. In contrast, ZENTURIO uses a directive-based language called ZEN [20] to annotate arbitrary files. ZEN directives are used to assign value sets to so called ZEN variables (e.g. MPIRUN in Ex. 6.3, MPILIB in Ex. 6.4). A ZEN variable can represent any problem, system, or machine parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. The value set represents the list of interesting values for the corresponding parameter. There are four kinds of ZEN directives. The *ZEN substitute directive* uses a pre-processor-based string replacement mechanism to substitute ZEN variables with one element from its value set (see Ex. 6.1 and 6.2). To complement some of the ZEN substitute directive's limitations, a *ZEN assignment directive* can be used to insert an assignment statement in the code (see Ex. 6.3 and 6.4). By default, the cartesian product of the value sets of all ZEN variables is computed. The number of possible value set combinations can be limited by means of *ZEN constraint directives* (see Ex. 6.1 and 6.5). ZEN also supports the specification of performance metrics for arbitrary code regions through the *ZEN performance behaviour directives* (see Ex. 6.6).

A file/application annotated with ZEN directives is called ZEN file/application. A *ZEN Transformation System* (ZTS) generates all ZEN file instances for a ZEN file, based on the ZEN directives inserted (see Fig. 1). The SCALEA [27] instrumentation system, which provides a complete Fortran90 OpenMP/MPI/HPF front-end and unparser, is used to instrument the application for performance metrics. ZEN performance behaviour directives are translated to SCALEA directives and command-line options. An Experiment Generator computes the cartesian product of all ZEN file instances to generate the corresponding ZEN application instances. A detailed description of ZEN is given in [20].

ZENTURIO is designed as a distributed service architecture, depicted in Fig. 2. It consists of four services, distributed over several sites. A *site* is a host on the Internet where a service is installed and which can be remotely accessed.

The *User Portal* (UP) is a GUI-based client, which represents the entry point of the system. It commonly resides on the user's local machine. The UP consists of an *Experiment Preparation* (EP) portlet which supports the preparation of an application for execution, and an *Experiment Monitor* (EM) portlet which submits and monitors experiments. Through the EP, ZENTURIO receives as input a ZEN application (including the application source and the set of input and output files), one compilation and one execution command, and the machine name on which to execute the experiments of interest.

These inputs are transferred to the *Experiment Generator (EG)* service at the G-Site. Based on the ZEN directives, EG automatically instruments the application and generates the set of application instances and the corresponding experiments (see Fig. 1). An *experiment* is an entity which is fully specified by an application instance, a compilation and an execution command, a compilation and an execution directory, and a set of output file names.

After the experiments have been generated, they are transferred to the E-site where an *Experiment Executor* (EE) service resides. The EE is a generic service responsible for compiling, executing, and managing the execution of applications on a target machine. EE interacts at the back-end with a scheduler. The current implementation of ZENTURIO supports PBS [30] for cluster and GRAM/DUROC [8] for Grid (Globus) applications. Transferring and compiling applications are optional tasks that can be omitted, for instance, when the client (site of user) and the target machines share a file system and the application has already been compiled. EE informs the UP (more precisely the EM) about important events during the life-time of an application through an asynchronous notification mechanism.

After each experiment has completed, a post-mortem performance analysis module of SCALEA is used to compute the requested performance overheads based on the raw performance data generated. The application output results and performance data can be optionally stored into an *Experiment Data Repository* (EDR). The user can visualise output results and performance data graphically by using the *Application Data Visualiser* (ADV) portlet.

A detailed description of ZENTURIO's functionality can be found in [21].

3. ZENTURIO WEB SERVICES

The main goals of ZENTURIO is to develop a distributed Grid architecture to support experiment management for cluster and Grid computing. Initially, we implemented the architecture based on the Jini [10] technology, which we describe in [21]. Recently, we decided to follow the emerging standard technology in designing Grid applications, namely Web services. In this section we present our experience in deploying our existing ZENTURIO services as Web services.

3.1 Web Services

In the past two years, a consortium of companies, working with W3C, have released a set of XML-based standards [17, 23, 6, 33, 29, 2] which represent the foundation of a new technology for programming B2B applications, called Web services [18]. A *Web service* is an interface that describes a collection of operations that are network-accessible through standard messaging. The Web Services Description Lan-

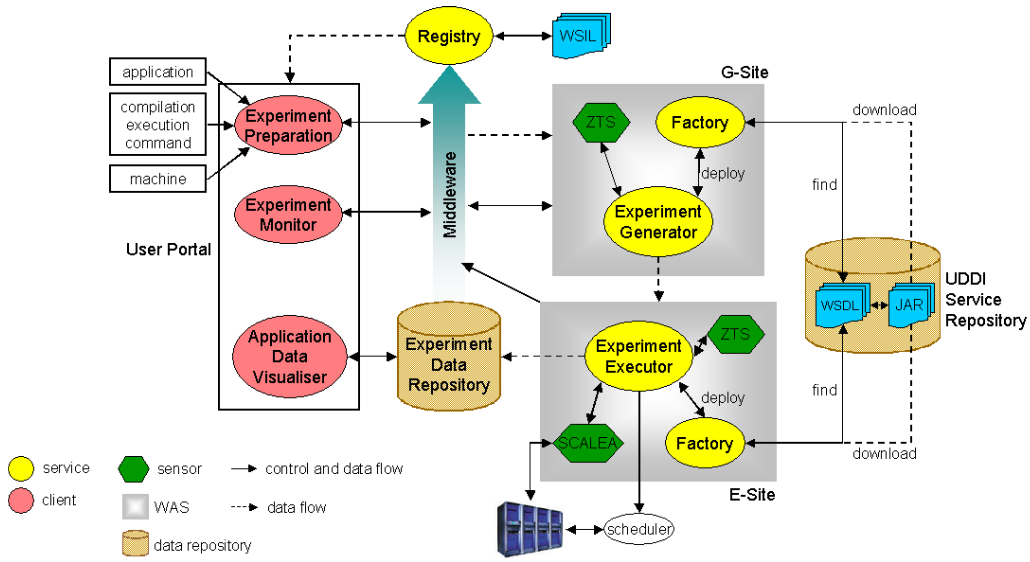


Figure 2: The architecture of ZENTURIO.

guage (WSDL [6]) is the de-facto XML-based standard for describing Web services. A running implementation of a Web service is called *Web service instance*.

A WSDL document is commonly divided into two distinct parts [18]: the service interface and the service instance. The *service interface* describes a Web service. It is the abstract and reusable part of a service definition, analogous to an abstract interface in a programming language. It can be instantiated and referenced by multiple service implementations. A service interface consists of the following XML elements: (1) `wsdl:types` contains the definition of complex XML Schema Datatypes (XSD) [33] which are used by the service interface; (2) `wsdl:message` defines the data transmitted as a collection of logical parts (`wsdl:parts` – e.g., input arguments, return argument, and exception messages), each of which is associated with a different type; (3) `wsdl:operation` is a named end point that consumes an input message and returns an output and a fault message (corresponds to a Java class method); (4) `wsdl:portType` defines a set of abstract operations (corresponds to a Java interface definition); (5) `wsdl:binding` describes the protocol and data format for the operations of a `portType`. The *service instance* part of a WSDL document describes an instantiation of a Web service. A Web service instance is modelled as a `wsdl:service`, which contains a collection (usually one) of `wsdl:port` elements. A `port` associates one network endpoint (e.g. URL) with a `wsdl:binding` element from a service interface definition. A common practice is do define the service interface in a separate *abstract interface WSDL document* which is further included in the *instance WSDL document* through an `import` element.

SOAP (Simple Object Access Protocol) [23] over HTTP has emerged as the XML-based standard network protocol binding for exchanging messages between Web services.

We have chosen Systinet's Web Application and Services Platform (WASP) for Java [24] to implement and deploy the ZENTURIO Web services. Fig. 3 illustrates the WASP Web service runtime environment. Every ZENTURIO Web service is deployed within the WASP *Web Application Server (WAS)*, which is an HTTP server and a servlet engine that hosts Web services (also termed as hosting environment). A single WAS runs on each ZENTURIO site.

We implement each ZENTURIO Web service as one Java class. Upon the deployment of a Web service, a Web service instance is created with an associated instance WSDL document, automatically generated by WASP via a Java2WSDL tool. The WSDL document has exactly one `portType`, which is homonym with the Java class name. Each method of the Java class is mapped to one `portType` operation. The WSDL file has exactly one `service` with one `port` element, which defines a URL address for the SOAP binding of the `portType`.

We implement the Service Repository using the WASP Universal Description, Discovery, and Integration (UDDI) registry [29]. The Service Repository contains information about Web service implementations, and not about Web service instances (which are published in the Registry service, see Section 3.4). We manually publish the ZENTURIO Web services (as yellow and green pages data) in the UDDI registry. Briefly, a `businessService` UDDI element is a descriptive container, which is used to group related Web services. It contains a `bindingTemplate` element, which describes the information required to invoke the service. The `bindingTemplate` contains a pointer to a `tModel` element, which describes meta-data of a Web service. According to [7] the interface part of the WSDL document is published as a UDDI `tModel` and the instance part is published as a `businessService` element (as URLs). In contrast, we use the `businessService` element to publish service implementation information of a Web service (not service instance, which is published in the Registry service, see Section 3.4). The `accessPoint` element of a `bindindTemplate` is set with the URL of the JAR package that implements the service.

We use a *direct publication* of the WSDL service interface part from service providers to service requesters through the UDDI registry. ZENTURIO clients (i.e. the UP) employ a *build-time dynamic binding* model [4] to connect to Web services (see Fig. 3). Based on the WSDL documents received, clients dynamically generate generic service proxies to communicate with the Web services. The Web Service Invocation Framework (WSIF) [9] enables the usage of the same proxy with any SOAP implementation and with any network protocol. Locating the service instance is a function supported by

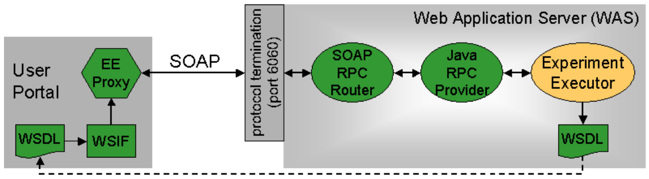


Figure 3: The Web service runtime environment in ZENTURIO.

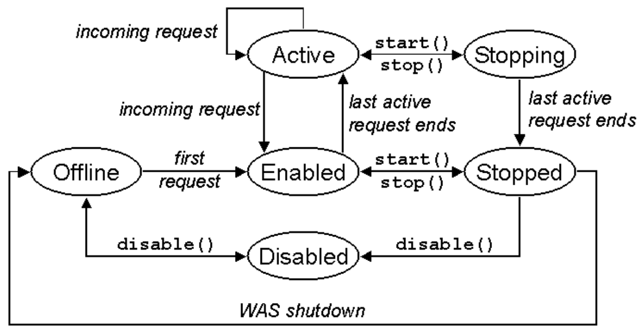


Figure 4: The state transition diagram of ZENTURIO Web services.

the Registry service (see Section 3.4). Each remote call from a client to a Web service is mapped to one SOAP RPC message. Upon receiving this message at the network endpoint, a SOAP RPC router (servlet) unmarshalls the message and forwards it to a Java RPC provider. The Java RPC provider loads the Java class specified in the SOAP message (if not already loaded) and invokes the appropriate method. The results of the method are returned to the SOAP RPC router which marshalls and transfers them to the requesting client.

However, in contrast to popular belief, a Web service is neither required to carry XML messages, nor to be bound to SOAP or the HTTP protocol, nor to run within a WAS.

Figure 4 depicts the state transition diagram of a Web service, deployed within the WASP WAS. *Offline* is the initial state and indicates that the service is not in memory, but will be loaded by the Java RPC Provider (and transferred to state *Enabled*) when a request arrives. In the *Active* state the service is processing one or more clients. The state *Stopping* indicates that a request to stop the service was issued, but some requests are still processed by the service. A service in the state *Stopped* remains in memory but rejects all incoming requests. *Disabled* means that the service is not in memory and cannot receive any requests. Transitions between states are performed by WAS either automatically (transitions with italicised text), or through explicit calls to the WASP administration service (transitions with typewriter style text).

Figure 5 displays a hierarchical classification of the Web services implemented by ZENTURIO, which we describe in detail in the next sections. Each service is a specialisation of an abstract ZENTURIO Web service (ZW). ZW defines and partially implements common functionality needed by ZENTURIO Web services. It implements the Producer and Consumer interfaces, which are defined by the generic ZENTURIO event framework (see Sections 3.7 and 4). The Producer and Consumer interfaces define a common generic event data structure which is introduced in Section 3.7. Currently, ZENTURIO defines 5 Web services, namely Factory (see Section 3.3), Registry (see Section 3.4), EG (see Section 3.5), EE (see Section 3.6), and Filter (see Section 3.8).

ZENTURIO services can be persistent or transient. Factory and Registry are persistent services while the others are transient. Persistent services are deployed once and made offline when the WAS shuts-down or starts-up. All ZENTURIO services can be *accessed concurrently* by multiple clients, which is an essential feature in a Grid environment.

3.2 ZENTURIO Web Service (ZW)

At the top of the service hierarchy (see Fig. 5) is a generic abstract service called ZENTURIO Web Service, which reflects common functionality shared by all ZENTURIO services. All ZENTURIO services are specialisations of ZW. The generic operations of ZW are the following: (1) retrieve the URL of the WSDL file; (2) set and control the service state within the WAS (see Fig. 4); (3) retrieve and set the soft-state termination time; (4) initialise the service after the transition from state Offline to state Enabled; (5) reset a service by eliminating all state information when the service is changing to the Disabled state; (6) retrieve the load (given in percentage) of a service. Operations 4, 5, and 6 are abstract and must be specialised by each ZENTURIO service. Explicit service termination can be achieved by providing a service termination time which is equal or prior to the current time. Destroying a ZW means to undeploy it from the WAS. We implement a lighter version of this method by changing the state of the service to Disabled. A subsequent recreation of the service uses the existing disabled instance and changes its state to Offline, thus saving expensive deployment/undeployment overhead.

3.3 Factory

Each WAS contains by default one persistent Web service, namely the Factory service. This service implements the *factory* abstract concept or pattern which is responsible for deploying existing Web services (as Web service instances) inside the WAS in which the Factory resides. The Factory searches the (UDDI) Service Repository for a service of a given type (as `businessService` name [7]). If a service is found, then the Factory accesses the associated URL of the service implementation and downloads the corresponding JAR package. We use the WaspPackager and deployTool [24] to deploy our Web services into the WASP WAS. After the service has been deployed, the Factory returns the URL to the instance's WSDL file. Clients use this URL to retrieve the WSDL file and dynamically bind to the service through run-time generated proxies. Before searching for a service, the Factory examines whether an instance of the same type has been previously destroyed and disabled (see Section 3.2). If such an instance is found, the Factory makes it offline again (see Fig. 4), thus saving expensive download, package, and deployment overhead. The ZENTURIO UP uses the Factory to create EG and EE service instances on the Grid sites where the user intends to generate and execute experiments.

3.4 Registry

The Registry service is a persistent service which maintains an updated list of the URLs to the WSDL files of the currently deployed services. It may reside anywhere in the system and there can be an arbitrary number of Registries for registering Web services. The Registry grants *leases* to registered services, similar to the Jini [10] built-in leasing mechanism. If a service does not renew its lease before the lease expires, the Registry deletes it from the service list. This is an efficient way to cope with network failures and

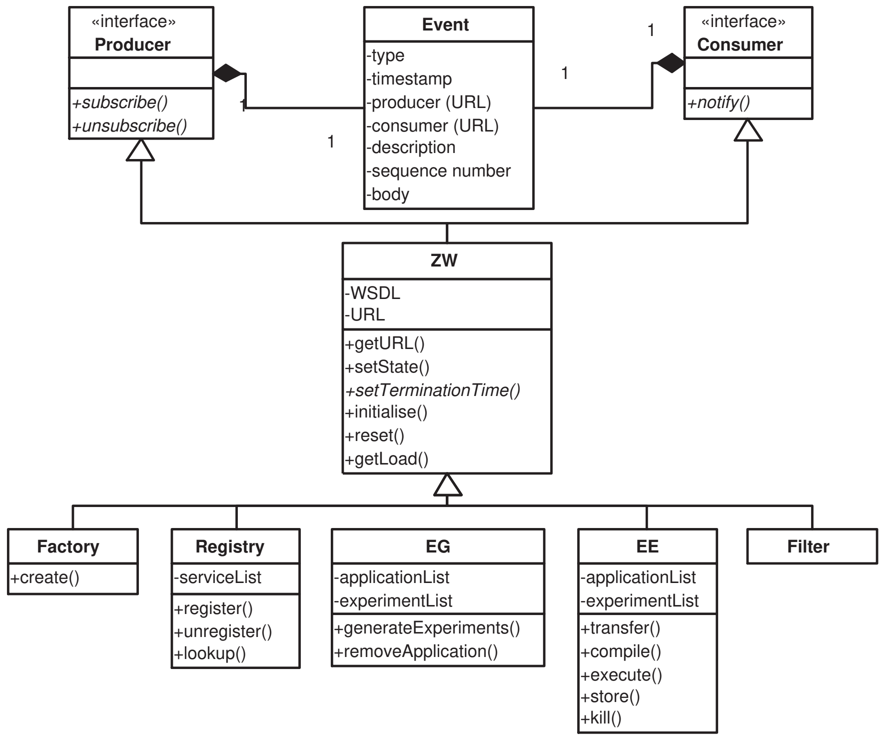


Figure 5: The ZENTURIO Web service classification.

insures network resilience in ZENTURIO. A leasing time of 0 seconds explicitly unregisters the service. An event mechanism is used to inform clients (e.g. UP) about new services that registered with the Registry, or when services did not renew their lease. Thereby, clients are always provided with a dynamically updated view of the Web service environment. The Registry is a generic service that operates on ZW services only and, therefore, can be used to register and discover services of any type. Lookup operations can be invoked against the Registry service based on the service URL (white pages), service (Java) type (yellow pages), or WSDL description (green pages).

The Web Services Inspection Language (WSIL [2]) defines a distributed Web service discovery method, which is complementary to the centralised service discovery method defined by UDDI. A WSIL document is an XML file containing references to Web services, which are URLs to instance WSDL documents. Each Registry generates one WSIL document containing references to its Web service instances. Currently, the ZENTURIO Registry does not generate such a WSIL document due to the lack of high-level tools in WASP. We are considering to use the WSIL4J (WSIL for Java) package from the IBM Web Services Toolkit (WSTK) [12], which was recently proposed to be included into Apache Axis [15].

At the moment there is a fixed number of Registries in the system, each one having a distinct pre-defined URL. The UP binds to all existing Registries at start-up.

3.5 Experiment Generator (EG)[1]

The EG uses ZTS to parse all ZEN files of a ZEN application and to generate all application instances and the corresponding experiments as described in Section 2 (see Fig. 1). After the experiments have been retrieved by the EE, the user can explicitely ask the EG to remove the ZEN application and the experiments associated.

3.6 Experiment Executor (EE)[2]

The EE Web service is responsible to execute and control experiments on the E-site. It can operate in five different modes, depending on the scheduler with which it interacts at the back-end: (1) *fork*, for single processor machines; (2) *PBS* [30] for workstation clusters; (3) *GRAM-fork* [8] for running an experiment with the Globus metacomputing infrastructure on a single Globus site, using fork as job manager; (4) *GRAM-PBS* for running an experiment with Globus on a single Globus site, using PBS as a job manager; (5) *DUROC* for running an experiment on multiple Globus sites. For clusters, the EE physically resides on the front-end node, while for Globus it can reside on any Globus site. We implemented the GRAM job submission mechanism on top of Java CoG Kit [31].

The EE was designed as a stand-alone Grid application and job management service independent of the ZENTURIO architecture. Thus, it can be deployed for other experiment management infrastructures as well. The EE provides functionality for: (1) adding/removing experiments; (2) compiling experiments; (3) executing experiments; (4) retrieving status information of experiments; (5) subscribing for status notification callbacks, which avoids permanent polling of clients (e.g. UP); (6) terminating experiments; (7) transferring experiment output (i.e. standard output, standard error, any output file, and performance data); (8) retrieving all experiments associated with a certain application (optionally being in a certain state). These methods can be applied

[1] `http://gescher.vcpc.univie.ac.at:6060/EG/`

[2] `http://gescher.vcpc.univie.ac.at:6060/EE/`

to individual or collective (i.e. all experiments belonging to an application) experiments. All EE methods support synchronous (blocking) and asynchronous (non-blocking) modes. In order to enable highly responsive clients, *asynchronous* methods can optionally invoke responses via notification callbacks (see Section 4, Tab. 2). This functionality is particularly useful for the EM as it avoids expensive polling for information.

3.7 Events

ZENTURIO services that require to send or receive events must respectively implement the Producer and Consumer interfaces. An *event* is a timestamped piece of data generated by a sensor and sent by a producer to a consumer. A *sensor* is an entity (e.g. instrumented code) that generates timestamped monitoring events. Sensors can be stand-alone or embedded inside a producer. An event *producer* is a Web service that implements the Producer interface and uses sensors to generate events. Further reusability could be achieved by decoupling the sensors from the producers through a well-defined interface. An event *consumer* is a Web service (usually a thread, part of a client application) that implements the Consumer interface. The Registry service stores information about existing producers and consumers.

We designed the event framework of ZENTURIO by closely following the Grid Monitoring Architecture (GMA [25]) proposed by the Grid Performance Working Group of the Global Grid Forum. Whereas GMA mostly focuses on performance events, ZENTURIO targets generic events.

Our event representation combines the two approaches addressed by GMA. We applied subtle modifications which we believe makes the specification clearer. An event consists of two parts. (1) The event *header* is the standard part of the event structure that comprises the following fields: (i) *event type* is an identifier that refers to a category of events, defined by an *event schema*; (ii) *timestamp* indicates when the event was generated; if events are buffered, then elements in the event body may contain additional timestamp information; (iii) *event producer, consumer (URLs), description, sequence number, and expiration timestamp* are optional fields; (2) The event *body* represents the effective information carried by the event. It consists of a *container* of elements where every *element* refers to a single event. The structure of an element (i.e. the type) is defined by an event schema. An element description, the measurement unit, and the accuracy can be optionally added for an element. We have a unique definition for all ZENTURIO events, which defines the body as a generic Java `Object`. The consumer casts it to the appropriate type based on the event type.

An *event description* may be also specified by a consumer when subscribing for an event. It consists of event parameters and filters. An *event parameter* specifies the properties (characteristics) of events to be sent to the user (e.g. process identifier for which to send process status events). Event parameters describe part of the event structure and are therefore included the as part of the event schema (as data members). A *filter* specifies under which conditions an event must be sent to the user (e.g. minimum value for a CPU load event).

ZENTURIO supports three kinds of interactions between producers and consumers, as specified by GMA. (1) *publish/subscribe* (PS) is a generalisation of the push model, in the sense that the initiator can be either the producer or the consumer. It searches in the Registry service for the other party (producer or consumer) and registers for (production or consumption of) events. The producer sends events to consumers until the initiator unsubscribes. (2) *query/response* (QR) generalises the pull model. The initiator is the consumer and the event is sent in a single response any time after the event has been requested. (3) *notification* (N) is a slight specialisation of the push model. The producer transfers the events to the consumer in a single notification with no preliminary subscription.

ZENTURIO implements the PS interaction by using the push model, i.e. the initiator is always the consumer. Consumers subscribe for events of producers (ZENTURIO services) by specifying the following inputs: the event type, an event consumer (URL of WSDL file), and optionally event parameters and filters, and the subscription expiration time. Events are sent from producers to consumers through an N interaction (no subscription) when important errors occur. We implement the QR interaction by invoking synchronous calls to Web services.

WASP Web services are likely to be heavy weight and necessitate deployment inside a WAS, which may not be available at every client site. For this reason we decided to use the light-weight XEVENTS [11], package to implement the PS and N interactions. XEVENTS provides a simple event framework built on top of a reliable application level messaging package called XMESSAGES. Producers and consumers are implemented by ZENTURIO as simple XEVENTS event publishers and message sinks, respectively. Since event persistence and reliability are not among our requirements, we do not make use of any advanced features of XEVENTS, such as event channels or publisher agents.

3.8 Filters

Filters can be either encoded inside the event producers (ZENTURIO services) or designed separately, as special kind of intermediaries. An *intermediary* is a Web service that insinuates between a producer and a consumer during an event notification. An intermediary can be shared by multiple producers and consumers. A *filter* is an intermediary which delivers to consumers a subset of the messages that it receives from producers. The event subscription method to ZENTURIO producers can receive as input (along side event type, consumer, and event parameters) an array of filters of the ZW type. The filters are chained such that the first filter receives messages directly from the producer and the last one delivers messages to the consumer. This general method of chaining arbitrary filters can be employed at certain latency costs (unless the deviated path through the filter has a higher bandwidth than the direct path producer-consumer). We have implemented an abstract template filter, which specialises ZW and has exactly one producer and one consumer. Our implementation is based on XEVENT's event publishers and message sinks; it could be extended with event channels and agents that provide both event persistence and reliability. Filters can be pluged-in by specialising this abstract class and implementing the filtering algorithm inside the notify method.

4. ZENTURIO EVENTS

Figure 6 depicts a generic event classification based on several event categories, which is targeted by ZENTURIO in a longer term. Table 1 displays the event categories currently supported by ZENTURIO. A dash means that the corresponding producers or sensors are not yet implemented.

Event Category	Producer	Sensor
Application Status	EE	ZTS, EE, EG
Process Status	EE	ZTS
Thread Status	EE	ZTS
Network Status	—	—
Site Status	—	—
Service Status	ZW, Registry	ZW, Registry
App. Performance	EE	SCALEA
Process Performance	—	SCALEA
Thread Performance	—	SCALEA
Network Performance	—	NetLogger
Site Performance	—	vmstat, iostat
Service Performance	ZW	ZW

Table 1: The event implementation support in ZENTURIO.

Some application status events are generated by the ZTS run-time instrumentation sensor (see Section 4.1). Application performance events are generated by the ZTS sensor which is based on SCALEA (see Section 4.2). The other event sensors are encoded inside the ZENTURIO Web services.

The UP is the only event consumer in ZENTURIO. A detailed description of the events supported by the ZENTURIO Web services is given in Table 2.

We introduced the ZEN language in Section 2 and described it in detail in [20]. Briefly, ZEN is a directive based language for the specification of application parameters and performance metrics. We enhance the ZEN language for application specific events, which is described in the following.

4.1 ZEN Application Event Directive

By using the *ZEN application event directive*, a user can define events in ZEN files. The ZEN application event directive has the following syntax:

zen-event	**is**	`EVENT` *ident* [`FILTER` *bool-expr*]
ident	**is**	*string*

The ZTS sensor replaces the ZEN application event directive with instrumentation code, which generates an event whenever the directive is reached by the executing program.

The event identifier *ident* is an arbitrary string that must be unique within a ZEN file. Optionally, the `FILTER` clause filters the events to those which satisfy the associated boolean expression. *bool-expr* represents a boolean expression (see [20]) defined over program variables which must be valid at the point where the ZEN event directive is defined. The syntax of the boolean expression is the one defined by the ZEN constraint directive [20], with the difference that it contains program variables instead of ZEN variables (along side constants). An eventual "variable not found" compilation error will be generated as an N (notification, see Section 3.7) event, by a subsequent file compilation process.

Example 4.1

```
!ZEN$ EVENT N1000 FILTER N > 1000
```

Ex. 4.1 defines a ZEN event directive, which generates an event of type `N1000` if the program variable N is greater than 1000.

4.2 ZEN Performance Behaviour Directive

ZEN supports the specification of performance metrics to be measured for specific code regions through the ZEN performance behaviour directive [20]. We enhanced the ZEN behaviour directive to support performance events, as follows:

global-perf	**is**	`CR` *cr_mnem-list* `PMETRIC` *pm_mnem-list* [`EVENT` *ident*] [`SAMPLE` *rate*] [`FILTER` *bool-expr*]
local-perf	**is**	`CR` *cr_mnem-list* `PMETRIC` *pm_mnem-list* [`EVENT` *ident*] [`SAMPLE` *rate*] [`FILTER` *bool-expr*] `BEGIN` *code-region* `END CR`

A performance behaviour directive collects performance metrics specified by the `PMETRIC` clause for all code regions indicated by the `CR` clause of the directive. Overall ZEN supports approximately 50 code regions (e.g. `CR_P` = entire program, `CR_L` = all loops, `CR_OMPPA` = all OpenMP parallel loops) and 40 performance metric mnemonics (e.g. `ODATA` = data movement, `OSYNC` = synchronisation, `ODATA_L2` = number of level 2 cache misses) for various programming paradigms, including OpenMP, HPF, and MPI. A complete list of mnemonics supported by ZEN can be found in [26].

The ZEN behaviour directive has been extended with three optional clauses to support application performance events:

`EVENT` defines the event type;

`SAMPLE` is an event parameter which defines the rate at which the performance metrics specified by the directive are periodically sampled. No sampling is done if this clause is missing. The measurement unit is times per second. For each measurement, an event of type *ident* is generated if the boolean expression specified by the `FILTER` clause yields true (or misses). The sampling rate also defines the expiration time of the events generated;

`FILTER` defines a filter as a boolean expression over the performance metric mnemonics specified by the directive (see Section 4.2). The performance mnemonics in the expression must be specified within the `PMETRIC` clause. An expression evaluation occurs at run-time at a rate specified by the `SAMPLE` clause. During evaluation, each mnemonic is instantiated with the corresponding performance data measured by SCALEA. If the expression evaluation yields true, an event of type *ident* is sent to the user.

Example 4.2

```
!ZEN$ CR CR_P PMETRIC ODATA, WTIME EVENT global
           SAMPLE 4 FILTER ODATA > WTIME/2
```

Ex. 4.2 shows a ZEN performance behaviour directive, which measures the global execution and communication time for the entire program. The two metrics are sampled 4 times per second. An event called `global` is generated if the communication dominates (is greater than half of) the overall execution time.

5. WEB SERVICES VERSUS OGSA

Recently, a new Open Grid Service Infrastructure (OGSI) working group has been set-up as part of the Global Grid Forum, to review and refine the Open Grid Service Architecture (OGSA) [14, 28], developed within the Globus project. OGSA defines an extension and a refinement of the Web service architecture, by specifying mechanisms for creating,

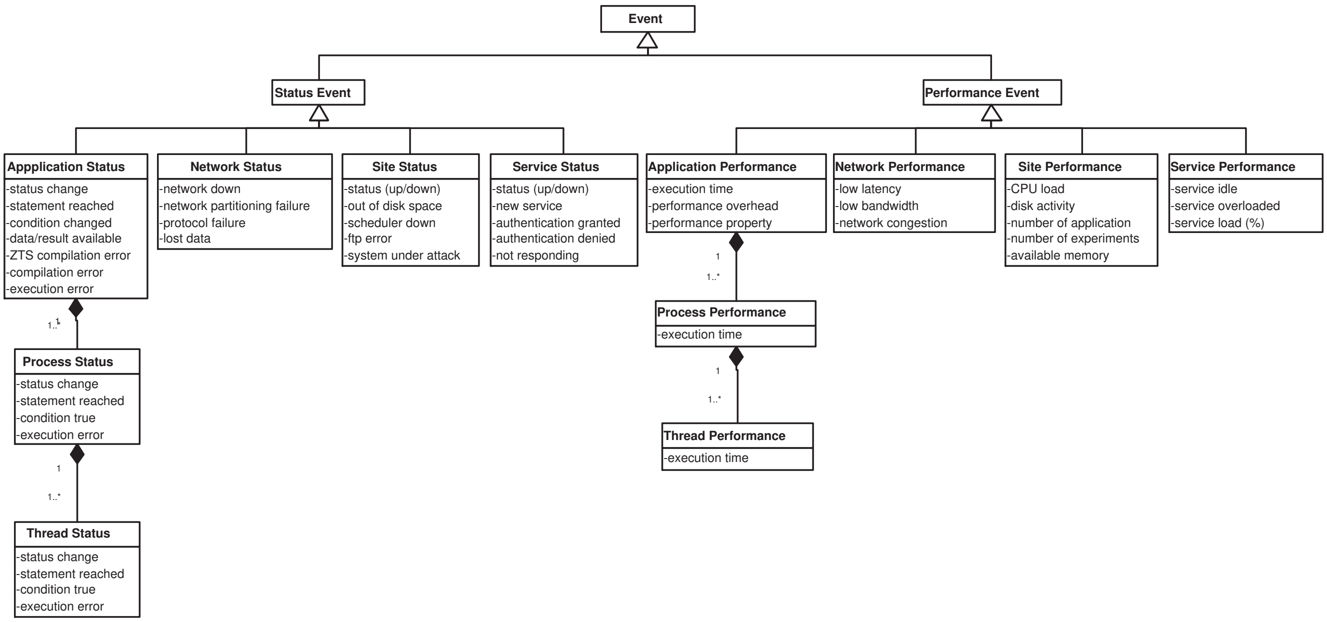


Figure 6: General event classification targeted by ZENTURIO.

Event Type	Producer	Consumer	Sensor	Parameters	Filters	Elements	Interaction
service state	Registry	UP	Registry	type, site		status (up/down)	PS, QR
new service	Registry	UP	Registry	type, site		service type, URL	PS, QR
out of disk space	EG	UP	ZTS, EG			site	N
ZTS compile error	EG	UP	ZTS			ZEN file, message	N
fie transfer error	EG, Factory	UP	EG			site	N
condition true	EE	UP	ZTS Instr.	identifier, buffer size	bool-expr		PS, QR
application performance	EE	UP	SCALEA Instrum.	metric, expiration, buffer size	bool-expr max, min, avg.	value	PS, QR
experiment status	EG, EE	UP	EE	ZEN app, ZEN vars		experiment, status	PS, QR
scheduler down	EE	UP	EE			site, sched. type	N
scheduler not supported	EE	UP	EE			site, sched. type	N
service load	EE, EG	UP	EE, EG	site, sample rate	max. load	no. apps, no. exps.	PS, QR
compilation error	EE	UP	EE			experiment, message	N
execution error	EE	UP	EE			experiment, message	N
experiment store error	EE	UP	SCALEA			experiment, message	N
ED access error	EE	UP	EE, SCALEA			site, message	N

Table 2: Overview of supported ZENTURIO events.

managing, and exchanging information among so called Grid services. A *Grid service* is defined as a Web service that confirms to a set of conventions (interfaces and behaviours) that specifies how it interacts with a client. An open-source preview release of OGSA is already available under the Globus toolkit public license. In this section we analyse the OGSA as a optional implementation technology for ZENTURIO and compare it against Web services.

There are lots of similarities between our service hierarchy and the OGSA `portType` specification. However, since our implementation is not (yet) based on OGSA, we did not update our terminology. ZW corresponds to the generic GridService `portType`, while the Factory and the Registry service implement the `portType`s with the same name. The GMA-based ZENTURIO event framework could be seen as a layer on top of the OGSA notification mechanism. The Producer and Consumer interfaces correspond to the NotificationSource and NotificationSink `portType`s, respectively. An event represents an OGSA notification message sent from a NotificationSource to a NotificationSink `portType`. OGSA does not make any difference between event parameters and filters, which are all mapped to OGSA notification constraints. We believe that the difference is purely semantic and therefore out of the scope of a low level architecture like OGSA.

The OGSA specification of `portType`s is visible in ZENTURIO at the Web service level (see Fig. 5). However, after WSDL documents are automatically generated by WASP during service deployment, the information about `portType`s is lost. Each ZENTURIO Web service is defined by one single `portType`, whose `name` attribute indicates the (Java) type of the Web service. The methods of each super-class and interface are automatically mapped to the `portType` of the service specialises or implements them. This WSDL definition does not confirm with the OGSA, since the `portType` information of super-classes and interfaces is lost. For instance, the notify method of the Consumer (NotificationSink) interface is defined within the `portType` of the service that implements this interface (e.g. Factory, Registry, EG, EE, Filter). We could see the (unique) `portType` of a service as representing the OGSA `serviceType` extensibility element which indicates the service type.

Until now we maintain only one implementation of the ZENTURIO services. Therefore, the implementation semantics addressed by the OGSA `serviceImplementation` and `compatibilityAssertion` extensibility elements are not (yet) relevant for our architecture.

The WSDL service instance part of a ZENTURIO Web service instance is represented by one `service` element, which contains one single `port` element (named as the Java class name) that instantiates the `portType` of the service. The `soap:address` binding element is used to associate the port with a URL address (i.e. a URI which corresponds to the HTTP SOAP binding). This URL corresponds to the OGSA Grid Service Handle (GSH), which uniquely identifies a Grid service in a Grid environment. In OGSA the WSDL document for one Grid service instance is called Grid Service Reference (GSR). Therefore, the OGSA recommendation of storing the GSH inside GSR is also fulfilled. In the WASP WAS, an HTTP GET operation on the GSH returns the GSR (the WSDL document) of a Grid service instance. This way does not fully confirm with the OGSA which requires the `.gsr` suffix to be appended to the GSH.

For example, our working EE service instance has the GSH: `http://gescher.vcpc.univie.ac.at:6060/EE/`. An HTTP GET operation with a normal browser displays the WSDL document of the deployed EE Web service instance. `http://gescher.vcpc.univie.ac.at:6060/admin/console` contains a portal to our WASP WAS, which contains the ZENTURIO services currently deployed (push the `Refresh` button). Under the constraint of having one single `port` in a service, the OGSA `instanceOf` extensibility element, used to describe a Grid service instance, is represented by this `port` element.

An important extension that OGSA adds to WSDL is the opportunity to specify transient state service information, mainly through the `serviceData` extensibility element. The WSDL document for a ZENTURIO Grid service instance contains static service metadata only, which is seen as one of the main Web services' limitations. Dynamic properties of a service instance, which correspond to the OGSA dynamic, instance specific `serviceData` WSDL extensibility elements can be obtained through specific RPC-based operations. An introspection mechanism to query the dynamic state of a service is not implemented by ZENTURIO Web services. Each service provides operations to access its private data members using an EJB [22] naming scheme. Additionally, each method is also replicated by a polymorphic FindServiceData method.

The OGSA HandleMap `portType`, which defines a second option for mapping one GSH to one GSR, is implemented by the ZENTURIO Registry service as a white-page lookup operation (see Section 3.4). Since GSH and GSR are already two references to a Grid service instance, we believe there is no need for the PrimaryKey `portType`.

6. EXPERIMENTS

In this section we illustrate how to use the ZENTURIO experiment management system for a performance study on the Grid under the Globus infrastructure. The performance study has been conducted for LAPW0, which is a material science kernel, part of the Wien97 package [3], that calculates the potential of the Kohn-Sham eigen-value problem. LAPW0 is implemented as a Fortran90 MPI program, which we run on a cluster of SMP nodes installed as a single Globus site. Shared memory communication is used inside SMP nodes. Experiments were submitted using the Globus Resource Allocation Manager (GRAM [8]) over PBS [30]. We have varied several application parameters by means of ZEN directives. The *problem size* is expressed by pairs of `.clmsum` and `.struct` input files, indicated in the `lapw0.def` input file (see Ex. 6.1). ZEN substitute directives are used to specify the file locations. The ZEN index constraint directive pairs the input files, according to the four problem sizes examined: 8, 16, 32, and 64 atoms. The *machine size* is controlled by the `count` parameter in the Globus RSL script (see Ex. 6.2), which we vary through a ZEN substitute directive. Based on the number of processors specified for an experiment, GRAM allocates the proper number of SMP nodes. We examined the performance for two interconnection *networks*: Fast Ethernet and Myrinet. The shell script `script.sh` (see Ex. 6.3) assigns the path of the `mpirun` command to the MPIRUN environment variable and starts the application. The application Makefile (see Ex. 6.4) sets the MPILIB environment variable to the corresponding MPI library. The global constraint file (see Ex. 6.5) ensures the correct association between the network specific (Fast Ethernet and Myrinet) MPI libraries and the corresponding `mpirun` script. The overall execution time and the entire communication time were measured by using ZEN behaviour directives (see Ex. 6.6) and the SCALEA MPI wrapper library.

In total, 8 ZEN directives were inserted into 6 ZEN files. Based on the ZEN directives, a total set of 320 experiments were generated on the G-site by the EG service and the associated ZTS sensor. The experiments are then automatically transferred to the EE service on the E-site, compiled, and submitted to GRAM for execution. The EM portlet periodically updates a view that shows the status of experiments (e.g., compiling, queued, running, terminated, stored) as they progress. After each experiment terminates, the performance and output data is stored into the EDR. We use the ZENTURIO ADV (see Section 2) to automatically generate visualisation diagrams from the performance data stored into the EDR.

Example 6.1 (Problem size – lapw0.def)

```
4,'znse_6.inm','unknown','formatted',0
...
!ZEN$ SUBSTITUTE 125hour.clmsum = { 125hour.clmsum,
             1hour.clmsum, 25hour.clmsum, 5hour.clmsum }
8,'ktp_.125hour.clmsum','old','formatted',0
...
!ZEN$ SUBSTITUTE 125hour.struct = { 125hour.struct,
             1hour.struct, 25hour.struct, 5hour.struct }
20,'ktp_.125hour.struct','old','formatted',0
...
58,'znse_6.vint','unknown','formatted',0
!ZEN$ CONSTRAINT INDEX 125hour.clmsum==125hour.struct
```

Example 6.2 (Globus RSL Script – run.rsl)

```
+
(&
 (resourceManagerContact="gescher/jobmanager-pbs")
(*ZEN$ SUBSTITUTE count\=4 = { count={2:40} }*)
 (count=4)
 (jobtype=single)
 (directory="/home/radu/APPS/LAPW0/znse_6")
 (executable="script.sh")
 (stdin="st.in")
 (stdout="st.out")
 (stderr="st.err")
)
```

Example 6.3 (script.sh)

```
#!/bin/sh
setenv MPI_MAX_CLUSTER_SIZE 1
cd $PBS_O_WORKDIR
nodes = `wc -l < $PBS_NODEFILE`
#ZEN$ ASSIGN MPIRUN = { /opt/local/mpich/bin/mpirun,
                    /opt/local/mpich_gm/bin/mpirun.ch_gm }
$(MPIRUN) -np $nodes -machinefile $PBS_NODEFILE lapw0
```

Example 6.4 (Makefile)

```
#ZEN$ ASSIGN MPILIB={/opt/local/mpich/lib/libmpich.a,
                     /opt/local/mpich_gm/lib/libmpich.a }
LIBS = ... -lsismpiwrapper $(MPILIB)
...
$(EXEC): $(OBJS)
        $(F90) -o lapw0 $(OBJS) $(LIBS)
```

Example 6.5 (Global Constraint File)

```
#ZEN$ CONSTRAINT INDEX Makefile:MPILIB==script.sh:MPIRUN
```

Example 6.6 (Source file – lapw0.F)

```
!ZEN$ CR PMETRIC WTIME, ODATA
```

Fig. 7(a) shows the scalability of the application for all four problems sizes. The scalability of the algorithm improves by increasing the LAPW0 problem size (number of atoms). For 8 atoms (`125hour.struct=125hour.struct`) LAPW0 does not scale at all. This is partially due to extensive communication overhead with respect to the overall execution time (see Fig. 7(b)). For a 64 atoms (`125hour.struct=5hour.struct`) the application scales well up to 15 processors, after which the execution time becomes relatively constant.

The interconnection network does not improve the communication behaviour (see Fig. 7(c)) because all receive operations are blocking, which dominates the effective transfer of a relative small amount of data among processes.

7. CONCLUSIONS

Developing ZENTURIO has been driven by the need to support scientists and engineers in their effort to conduct parameter studies, performance analysis and tuning, and software testing for a wide variety of parallel and distributed systems, including cluster and Grid architectures. In this paper we describe our experience with developing ZENTURIO as a collection of Web services. Web services provide an ideal infrastructure to distribute tools in a Grid environment in order to improve reusability of various tool components, to locate other services, and to support seamless access to services by multiple clients simultaneously. Existing toolkits are rapidly maturing and substantially alleviate the deployment of (Java and C++) applications as Web services. We compared our Web-service based implementation of ZENTURIO with the Open Grid Service Architecture (OGSA) and highlighted similarities and differences. The functionality of the `portTypes` (GridService, Factory, Registry, NotificationSource/Sink, HandleMap) proposed by OGSA is largely provided by ZENTURIO Web services. However, there is no automatic support for generating WSDL documents out of ZENTURIO Web services, that include OGSA `portTypes`. The key extensions to Web services proposed by OGSA – not supported at the moment – include the service type and dynamic service state information. We will base our implementation on OGSA, as soon as a stable implementation will become available within the Globus 3.0 toolkit.

8. REFERENCES

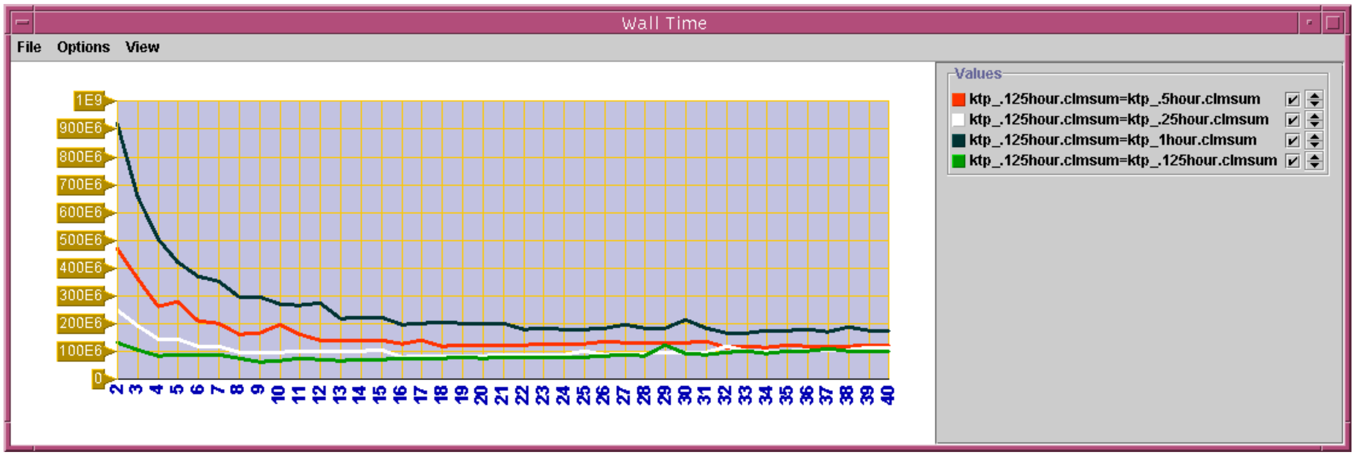
[1] D. Abramson, R. Sosic, R. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations high performance parametric modeling with nimrod/G: Killer application for the global grid? In *Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing (HPDC-95)*, pages 520–528, Virginia, August 1995. IEEE Computer Society Press.

[2] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies. Web Services Inspection Language (WS-Inspection) 1.0 . `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`.

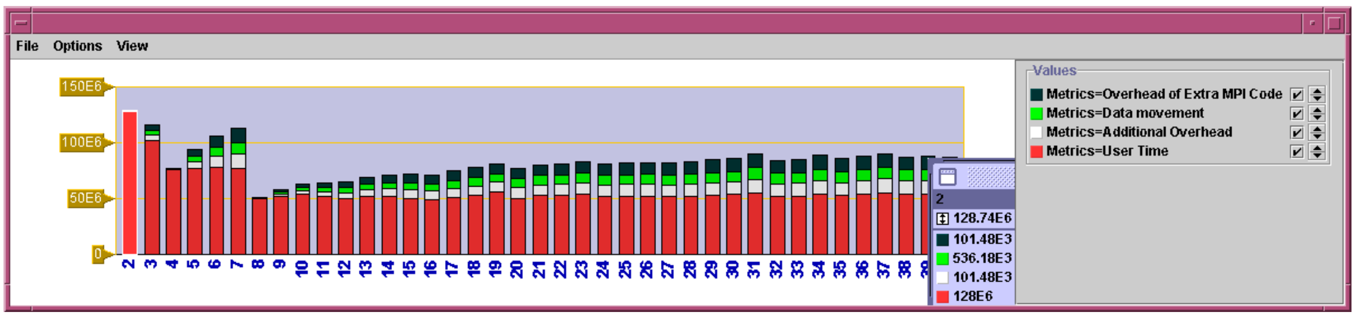
[3] Peter Blaha, Karlheinz Schwarz, and Joachim Luitz. *WIEN97: A Full Potential Linearized Augmented Plane Wave Package for Calculating Crystal Properties.* Institute of Physical and Theoretical Chemistry, April 2000.

[4] Peter Brittenham. Web Services Development Concepts (WSDC 1.0). Whitepaper, IBM Software Group, May 2001. `http://www-4.ibm.com/software/solutions/webservices/pdf/WSDC.pdf`.

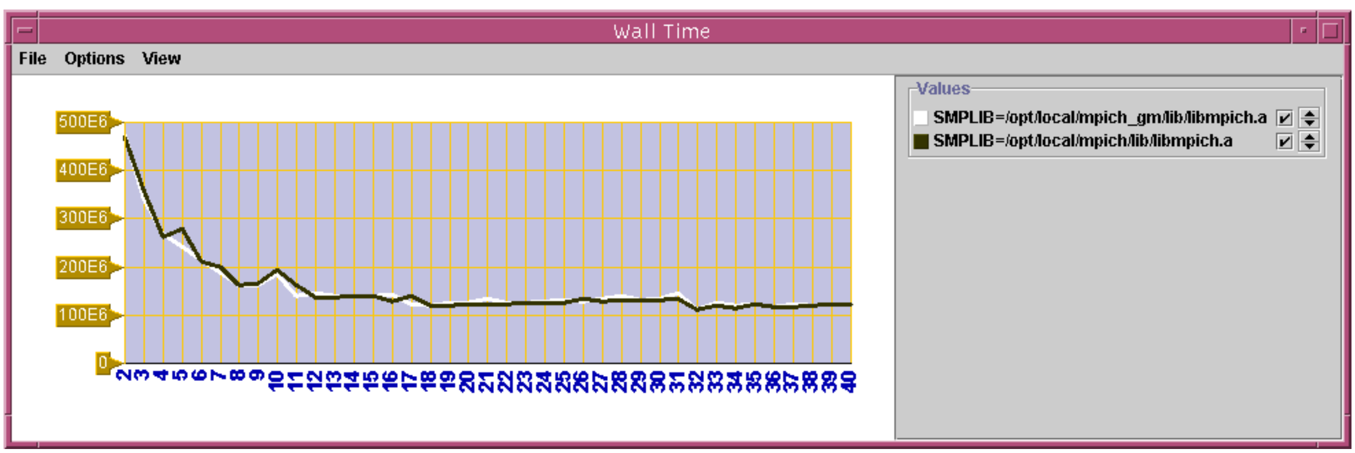
[5] Nat Brown and Charlie Kindel. *Distributed Component*



(a) Wallclock time for 4 Problem Sizes, Fast Ethernet.



(b) Overheads, 8 Atoms (`125hour.struct=125hour.struct`), Myrinet.



(c) Network Comparison, 64 Atoms (`125hour.struct=5hour.struct`).

Figure 7: ADV Diagrams for LAPW0 for varying number of processes. Experiments have been executed on an SMP cluster under the Globus infrastructure.

Object Model protocol: DCOM/1.0. Microsoft Corporation and Redmond, WA, January 1998.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL), March 2001. `http://www.w3.org/TR/wsdl`.

[7] Francisco Curbera, David Ehnebuske, and Dan Rogers. Using WSDL in a UDDI Registry 1.05. Uddi working draft best practices document, UDDI Organisation, june 2001. `http://www.uddi.org/pubs/wsdlbestpractices-V1.05-Open-20010625.pdf`.

[8] Karl Czajkowski, Ian Foster, Nick Karonis, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[9] Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski, and Sanjiva Weerawarana. Web Services Invocation Framework (WSIF). In *Proceedings of the OOPSLA 2001 Workshop on Object-Oriented Web Services*, Tampa, Florida, USA, October 2001.

[10] W. K. Edwards. Core Jini. *IEEE Micro*, 19(5):10–10, September/October 1999.

[11] Indiana University Extreme! Computing Group. Application Level Events and Message Service. `http://www.extreme.indiana.edu/xgws/xevents/`.

[12] John Feller. IBM Web Services ToolKit – A Showcase for emerging Web Services Technologies. `http://www-4.ibm.com/software/solutions/webservices/wstk-info.html`, April 2002.

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[14] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. The Globus Project and The Global Grid Forum, January 2002. `http://www.globus.org/research/papers/OGSA.pdf`.

[15] Apache Software Foundation. Apache Axis. `http://xml.apache.org/axis`.

[16] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2002. Designing and building distributed applications.

[17] Elliotte Rusty Harold. *XML: extensible markup language*. IDG Books, San Mateo, CA, USA, 1998.

[18] Heather Kreger. Web Services Conceptual Architecture (WSCA 1.0). Prepared for Sun Microsystems, Inc., IBM Software Group, May 2001. `http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf`.

[19] David S. Linthicum. CORBA 2.0? *Open Computing*, 12(2):68–??, February 1995.

[20] Radu Prodan and Thomas Fahringer. ZEN: A Directive-based Language for Automatic Experiment Management of Parallel and Distributed Programs. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP-02)*, Vancouver, Canada, August 2002. IEEE Computer Society Press. To appear.

[21] Radu Prodan and Thomas Fahringer. ZENTURIO: An Experiment Management System for Cluster and Grid Computing. In *Proceedings of the 4th International Conference on Cluster Computing (CLUSTER 2002)*, Chicago, USA, September 2002. IEEE Computer Society Press. To appear.

[22] Bill Roth. An introduction to Enterprise Java Beans technology. *Java Report: The Source for Java Development*, 3, October 1998.

[23] A. Ryman. Simple Object Access Protocol (SOAP) and Web Services. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 689–689, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[24] Systinet. Web Applications and Services Platform. `http://www.systinet.com/products/`.

[25] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. *A Grid Monitoring Architecture*. The Global Grid Forum, January 2002. `http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf`.

[26] Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International Europar Conference(EuroPar 2002)*, Lecture Notes in Computer Science, Paderborn, Germany, August 2002. Springer-Verlag. To appear.

[27] Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On Using SCALEA for Performance Analysis of Distributed and Parallel Programs. In *Proceeding of the 9th IEEE/ACM High-Performance Networking and Computing Conference (SC'2001)*, Denver, USA, November 2001.

[28] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. *Grid Service Specification*. The Globus Project and The Global Grid Forum, February 2002. `http://www.globus.org/research/papers/gsspec.pdf`.

[29] UDDI: Universal Description, Discovery and Integration. `http://www.uddi.org`.

[30] Veridian Systems. PBS: The Portable Batch System. `http://www.openpbs.org`.

[31] Gregor von Laszewski, Ian Foster, and Jarek Gawor. CoG kits: a bridge between commodity distributed computing and high-performance grids. In *Proceedings of the ACM Java Grande Conference*, pages 97–106, June 2000.

[32] W3C. Web Services Activity. `http://www.w3.org/2002/ws/`.

[33] W3C. XML Schemas: Datatypes. `http://www.w3.org/TR/xmlschema-2/`.

[34] M. Yarrow, K. M. McCann, R. Biswas, and R. F. Van der Wijngaart. Ilab: An advanced user interface approach for complex parameter study process specification on the information power grid. In *Proceedings of Grid 2000: International Workshop on Grid Computing*, Bangalore, India, December 2000. ACM Press and IEEE Computer Society Press.