2010 Sixth IEEE International Conference on e–Science Workshops

A JEE-based Architecture for Distributed Multi-Domain Resource Accounting

Michael Brenner, Jan Wiebelitz and Matthew Smith
Gottfried Wilhelm Leibniz Universität
Distributed Systems Security Group
Research Center L3S, Hannover, Germany
{brenner,wiebelitz,smith}@dcsec.uni-hannover.de

***Abstract*—Many different accounting systems currently exist for the European Grids and for other distributed systems in general. Even though they often share equivalent technical approaches, concerning accounting metrics and batch-system log access, there are lots of different data formats, accounting-database layouts, communication- and implementation details that hinder a consistent and comprehensive accounting of distributed resources. An eventual technical convergency is even more required, as the National Grid Initiatives move forward to constitute a European Grid that comprises the national infrastructures. We present EEGAS, a Java Enterprise based grid accounting system, that addresses the current problems. Our system makes use of standard building blocks, where possible, to achieve high scalability and application server console-based configurability. It serves as an integration point for existing accounting systems and uses the OGF-UR data format, which we use to generate a Java domain model and a database schema. We also present sensor components for CPU and storage accounting, resource capacity discovery, as well as different message- and service-based transport components that support a fault-tolerant operation of a distributed accounting network that features different administrative domains rather than a single centralized accounting authority.**

***Index Terms*—EEGAS, Distributed Resource Accounting, Capacity Discovery, Java EE, OGF-UR**

I. Introduction

The current landscape of distributed resource accounting is fairly spacious. When examining the different existing accounting systems, we find many similarities concerning the technical approaches. Often the data about resource usage is gained by accessing the log files of the underlying resource-systems. This information is then interpreted and correlated to meet the requirements of the different accounting metrics for CPU, storage or license usage. In contrast to the common conceptual aspects, we face large differences in the implementations. The most important problem is, that there is no unified data format; a fact that makes interoperability between different systems extremely tedious and even avoids it in most cases. As a direct consequence, we lack a common data format to exchange accounting data. The *Usage Record Format* that has been defined by the Usage Record Working Group of the *Open Grid Forum* has still not been widely accepted since the latest version.

Being at the beginning of the transition phase from the European EGEE project to the European Grid Initiative EGI, it is even more desirable to eventually speak a common language, which makes the exchange of accounting information feasible. It also allows building a multi-domain accounting network in which each domain controls its own information and propagates data to a higher administrative level, in a form that considers both, technical and legal issues.

Most existing accounting systems are sort of an evolutionary product that started with the requirement to only consider local clusters and batch systems. As a consequence, representative systems like the *Distributed Grid Accounting System (DGAS)* do provide a hierarchical infrastructure option [1] but use proprietary data exchange formats and proprietary relational database layouts [2] [3]. The *Accounting Processor for Event Logs (APEL)* makes use of a messaging system with unified data formats, but is a single domain system, which only allows one central authority [4].

To face the challenges of the converging European Grids we propose our accounting system EEGAS, that bases on the well defined Java Enterprise Architecture and the OGF-UR record format. We have designed a distributed accounting architecture that satisfies the identified requirements. Originally intended as an infrastructure extension to DGAS, EEGAS has evolved towards a comprising accounting solution with enhancements towards distributed resource discovery and support-functionality for meta-scheduling.

The paper structure is as follows: in Section II we discuss the Java Enterprise Architecture and the platform components that we will use to construct our application architecture. We briefly describe the function of each required part of the Java ecosystem and how it fits in the big picture. Section III defines our accounting system's elements and how they map on the Java architecture components. We discuss the data format, the transport mechanism, sensor technology and other functions, as a foundation for the following sections. The distribution model of our accounting system is presented in Section IV. We will show, how different requirements to the data-flow and physical component distribution can be satisfied using our system's elements. We describe sample configurations for common accounting scenarios with different domain models and sketch how distributed resource capacities can be tracked within these domains or by a central authority.

Section V gives a resource discovery scenario and briefly describes the required steps to integrate the functionality into

978-0-7695-4295-9/10 $26.00 © 2010 IEEE
DOI 10.1109/eScienceW.2010.15

IEEE computer society

a meta-scheduler.

Implementation details are discussed in Section VI. We provide information on special issues when using the accounting components and the underlying frameworks.

Related work and The conclusion is provided in Section VII.

II. The Java Enterprise Architecture

The *Java Enterprise Edition (JEE)* is a specification of an architecture and software platform that handles complex, transactional, component-based Java systems. This usually, but not necessarily comprises a web-based front-end and often a database backend. There are several free and commercial implementations of the JEE specification and almost any existing framework or component in the Java ecosystem can be connected to a JEE application, which reduces individual programming effort. This section gives a brief introduction to JEE and the typical components and frameworks as depicted in Figure 1.

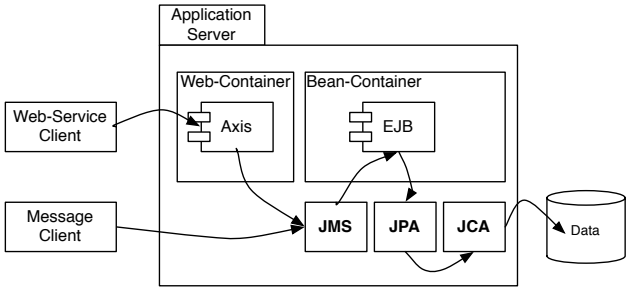

Fig. 1. Basic JEE-based Architecture

The *application server* is the main entity of any JEE installation. It serves as a central piece of a puzzle, to which different loose pieces are connected and form the infrastructure that's required for a particular application or set of applications. It essentially is a runtime environment or *container* for Java components, that obey the JEE specification (often called *enterprise beans*), and a number of interfaces that connect the JEE components to each other, to external components or systems and that provide typical services to components. The application server handles the life-cycles of the objects that belong to a component and it is also responsible for the management of transactions and object persistence. A distributed web application is often divided into three tiers, which are basically software layers that combine groups of actions required for a certain state of an application. In this sense, the enterprise bean container which is the core of the application server, is the perfect middle of this architecture (the mid-tier), because it handles the actual business logic of an application but also provides a server-side client tier (the web delivery function) and a rear end that connects to possibly different backend systems (the data tier). The mentioned *rear end* can also point to the front, when it is configured to handle certain client message types.

The web delivery function (tier 1) is a combination of a web server and a container for special java objects (*servlets*), that handle web traffic and generate web pages for client-side display. This sort of connection also applies to *web-service* connections that use servlet driven machine-generated inputs and machine-readable outputs. A servlet is a simple java object that has to implement a set of methods that allow the *servlet container* or *web container* to pass http requests and to receive http responses from it for delivery to the client. The JBoss application server is shipped with the *Apache Tomcat* web container, which comprises the *Apache Coyote* web server and the *Apache Catalina* servlet engine.

JEE specifies a couple of methods to allow mid-tier components to interact with backend-systems. One framework is the *Java Messaging Service (JMS)* that defines interfaces to a message-oriented middleware. This is a system that handles typical messaging patterns and modes. It is responsible for the reliable delivery of messages from a message producer to a message consumer. The consumer and one or more producers communicate by exchanging messages over a *message queue* that can sort and deliver messages according to different properties like priority or order, which can be configured. For our implementation, we apply the *Apache ActiveMQ* as a messaging provider that implements the JMS specification.

JEE defines a way to store application data and to save the state of an application by persisting the objects that represent the application. This requires an application to use *enterprise beans* that implement methods for a transparent database access. The *Enterprise Java Beans (EJB)*-specification claims the functionality of object-relational mapping, which is required to manage a relational database using the corresponding java domain model. The system, that we describe here, uses JBoss *Hibernate*, which is a reference-implementation of the EJB3-specification and the *Java Persistence API (JPA)*, that uses advanced Java-technologies like runtime class-reflection, dynamic SQL generation and annotations.

For access to other systems, JEE defines a set of interfaces in the *Java Connector Architecture (JCA)*. Connectors typically provide access to databases or other resources like mainframes or transaction monitors, but also to messaging, like mentioned before. The core function is to handle connections and transactions that are used to communicate with a remote entity. For this purpose the *resource adapter* implements three *system contracts* that specify the behavior of connections (1), transactions (2) and security (3) of the connected resource. In our accounting system, we use the JCA-spec to integrate a MySQL database system into the JEE-environment. For this purpose we provide a specific MySQL connector-library that matches the connector architecture on the server's end and applies to the *Java Database Connection specification (JDBC)* on the other side.

The JCA is designed to decouple i.e. the database system from other components running in the application container, like the persistence mechanism, and to use an abstraction that can be easily configured over the application server's administration console. The application server can also take

care of a connection pool which optimizes the network access to a resource. To achieve this, the resource adapters register their connections and resources to the application server and its *naming service*. The applications discover required resources by querying the *JNDI (Java Naming and Directory Interface)*, that implements access to the naming service.

III. Accounting System Components

This chapter defines our accounting system's elements and how they map on the Java architecture components. We give a detailed description of the system's parts, like the data format, the transport mechanism, sensor technology and other functions, as a foundation for the following chapters.

A. Transport Framework

The transport framework is one of the core features of our accounting system. By offering both, a web-service interface and message-based communication, we support a multi-channel strategy to access the accounting data repository. The web-service interface is a way to support lightweight clients or clients that come as embedded components in existing service-infrastructures. However, the focus of our transport-framework is on the messaging system.

Webservice Frontend

One channel to submit or to access accounting data is the web-service interface. Using this interface is very easy, since the appropriate stubs and data model components can be automatically generated from the provided WSDL-definition. This channel can be used for embedded service-based components as well as for the dedicated sensor components. So, the decision, which channel should be used in a given scenario, is up to the system architect.

Message Bus Components

The central transport mechanism in our architecture is a message bus (in other terms a *store-and-forward network federation)*, which consists of a number of distributed message queues in a broker network. It is implemented on top of the Apache ActiveMQ messaging middleware and handles all internal traffic on the servers and relay nodes. Additionally to the Axis-based web-service gateway, remote clients and relays can also send messages to the bus directly, using a message protocol. This approach unifies the data flow and allows easy support of a multi-channel strategy, which decouples the business logic from the data input channel. Furthermore, we provide message relay components that can be deployed on every site as network-concentrators and to order message transport and to support failover features. The *message relay component* is the most important element to build a hierarchical or networked accounting infrastructure. It requires a dedicated ActiveMQ installation, which is used to collect, order and forward messages, according to the local relay policy. The relayer also provides content-based message routing functions that are required to build a flexible infrastructure. The relayer acts as both, as a message repeater and multiplier, as well as a content-based routing element. To operate the relayer, a proper configuration has to be provided, that defines the inbound message queue and one or more outgoing queues. A sample configuration for a basic, unconditional forward operation is given in Listing 1.

```
inBroker  = tcp://srcbroker.com:61616;
            queue/clientQueue
outBroker = tcp://dstbroker.com:61616;
            queue/dgasInbound
```

Listing 1. Relay Configuration

The configuration parameters for each end of the communication channel contains the host address *(broker url)* and the queue name on that particular broker. On relay startup, a connection to both ends of the channel is established. ActiveMQ offers a failover feature that has two advantages: we can force an automatic reconnect in case an established channel gets disconnected, and we may provide a number of broker-URLs that build a failover group of distributed brokers. The basic failover-configuration for unconditional forwarding is shown in Listing 2.

```
inBroker  = failover:(
            tcp://srcbroker.com:61616);
            queue/clientQueue
outBroker = failover:(
            tcp://dstbroker.com:61616,
            tcp://dstbroker2.com:61616);
            queue/dgasInbound
```

Listing 2. Failover Relay Configuration

To configure the relay as a message multiplier, we just add outQueueNames and outBrokerUrls for every destination broker or every group of failover brokers.

Content-based Message Routing

An important function of the relay component is the ability to perform a content-based routing. This feature allows to decide, which next receiver is appropriate for a particular message, according to the contents or message headers. Content-based routing is only available for the unencrypted parts of a message that are readable by the relayer. For routing operation, we have to define a selection condition for each conditional destination. A selector is formulated as a regular expression that is applied on the message content and / or the message header. A routing condition with a message header selector and a message content selector is depicted in Listing 3.

```
outBroker =
  tcp://dstbroker1.com:61616;
  queue/dgasInbound;
  header(
    jmsConsumerIdentifier=broker1|broker2)
outBroker =
  tcp://dstbroker2.com:61616;
  queue/dgasInbound;
  content(
    gcwn[0-90-9]\.uni-hannover\.de)
```

Listing 3. Route Condition

The presented mechanisms, which are contained within the relay- and routing-component, support the construction of complex and flexible, distributed infrastructures. To complete the bus component overview, we describe the addressing schema for the messages.

Message Addressing Schema

To allow indirect addressing of components that can only be reached over relay hops, we have to define an addressing schema, that contains at least two receiver identities. The schema, that we use here refers to *Web-Service Addressing (WS-A)*. The first identity is that of the *ultimate receiver*. This is the accounting authority, which is responsible for a particular accounting record. The other identity is the *next receiver* in the sense of a relay hop. Thus, the next receiver changes during the travel to the ultimate receiver on every visited relay, according to the relay's routing policy. The receiver information is implemented using JMS properties, that are appended to the messages. However, all infrastructure components may or may not evaluate the addressing information and are allowed to override that information to comply with a local forwarding policy.

B. Sensors

Sensors are deployed on the distributed resources to record usage data. The sensors in our architecture feature a modular design with a *sensor core* and two plug-ins, one on each side. The *resource-side* plug-in gathers the required information on the resource, according to the various accounting metrics. The *bus-side* module handles the transport and can be web-service- or message-driven. Each tuple of a reportable measure and a transport-model requires a appropriate plug-in configuration for the sensor. A sample configuration for a Torque CPU accounting via the message system is shown in Listing 4.

```
adapterClass=de.dgas.ee.sensors.torque.Log
parserClass=de.dgas.ee.sensors.torque.Parser
connectorClass=de.dgas.ee.sensors.connector.Mq
```

Listing 4. Sensor Configuration

The adapter class defines the method to access the accounting information. In the sample case this is the Torque log file which can be easily accessed with a file reader. The parser class is the module that implements the handling of a specific content structure. In particular, the parser plug-in has to produce a valid OGF record from the batch-system specific accounting record. To achieve this, it makes use of the Java data-mapping framework, that we have implemented and that is described in section III-E. The connector class points to the communication handler module. The sample shows the selection of the message-based communication handler. In order to report real-time resource allocation figures to the authority, the sensor is operated in *early record release* mode which reports the resources that are assigned a particular job in a *pending* record, only containing the job start time. Resources with a pending job end time are considered *occupied*.

C. Accounting Authority

Located in the technical center of an accounting domain is the *message core*. Essentially this is a message queue with a connected message handler that receives accounting messages for its own domain, but can also be used to obtain data that has to be preprocessed before being forwarded to a higher level accounting authority. The result of preprocessing may be some sort of aggregate record that's supposed to conceal the composition of a distributed job or just to condense the information, while delivering the metric-dependent figures, like summarized CPU-time or memory usage. The message core is an integral part of the accounting authority component. We have implemented the accounting core by means of the *JEE message-driven enterprise-bean concept* for the following reasons. First, the integration into a complex infrastructure is simplified, because it's possible to deploy a message-driven bean into the bean-container. Second, the bean can use all the abstractions of resources that the container provides. Third, to face load issues, we are able to deploy a number of uniform message-beans that help scaling the application performance.

An accounting domain is defined by its central accounting authority (AA). This is the responsible entity for the accounting- and capacity-data of all connected sites. To deploy an AA, the installation of the application server along with a message broker is required. The AA grants full data access to the domain administrator and administrative processes, thus allowing data processing as required by local accounting- or economic policies. The AA installation also features the functions to preprocess data in order to meet the requirements of a higher-ranking authority and to provide information on available capacities in real-time. The processing model, that's implemented by the authority component, consists of three stages. Stage one is the preprocessing stage, where data manipulation and aggregation can be configured. The second stage is the redirector, which decides, if a message has to be re-routed after preprocessing and therefore is delivered back to the message bus, where the routing and forwarding policies are applied again. If the message proceeds to the third stage, then it is stored by the local database persistence mechanism. The structure is shown in Figure 2.

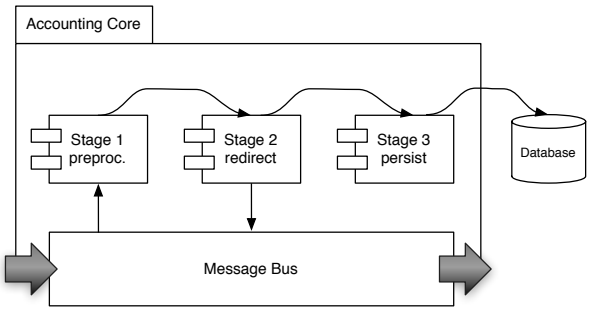

Fig. 2. Accounting Core Architecture

D. Database and Hibernation

EEGAS uses a relational database model which is a representation of the OGF-UR draft definition. The automatic generation is achieved using the XML-Beans framework which is also in charge of translating between the XML data streams and the java object model. The physical access to the relational database is encapsulated in the Hibernate-framework, which is providing OR-mapping and implements the Java Persistence API. The database connection is represented by a JDBC-datasource and handled by the application server's connection pool.

E. OGF Message Format

The Open Grid Forum has defined the record format to support sharing of distributed resources, where sites must be able to exchange basic accounting and usage data in a common format. The *OGF UR Format Recommendation* focuses on the representation of resource consumption data. The record format is intended to be specific enough to facilitate information sharing among grid sites, yet general enough that the usage data can be used for a variety of purposes: traditional usage accounting, charging, service usage monitoring, performance tuning, etc. The format recommendation outlines the basic building blocks of the accounting record, and how to properly represent it. Regarding the participating assets, the format definition bases on a couple of assumptions. The fundamental grid component is a resource and the fundamental consumer is a grid job. These jobs may be interactive or batch-driven and the usage recording is either detailed or in an aggregated form. The basic record consists of a number of *base properties* that represent most basic job accounting activities. Additionally there are *differentiated properties* that can hold the figures for metrics that occur more than once in one single record. To be able to model grid- and site-specific additions to the accounting record, the UR-schema defines *extensions*. Sites that exchange extended records must agree on a joint extension model [6].

EEGAS features a flat Java object model as a representation of the OGF-UR schema and a relational database model for persistence. These models have been automatically generated from the OGF-UR XML-schema. The transformation into the Java model is carried out by the XMLBeans framework. All processing on the accounting data is performed in the resulting Java object model. The wire format is the OGF XML-representation and this format is used in most cases from the sensor component to the backend database handler. So all components communicate via XML which makes it easier to integrate third-party components into our data flow, as long as they understand and produce the OGF format. The format transitions in a typical accounting workflow are shown in Figure 3.

The translation between the XML format and the database *data manipulation language (DML)* is not object-driven. This means, that in cases where no preprocessing is required, the database handler takes the XML record directly from the accounting authority and maps it on the database schema.

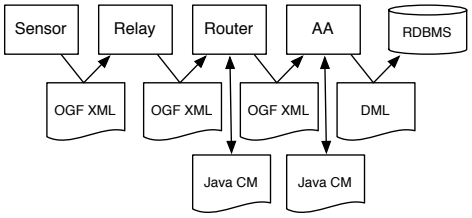

Fig. 3. Data Formats and Flow

If the AA has to process the record before persisting it, it transforms it from XML to the Java object model and back to XML before passing it to the database handler or the message handler. The situation on a router, that has to read and write addressing information of a record, is similar.

The direct mapping from XML to DML results in a series of SQL commands that are generated from an SQL template and the extracted strings from the XML record. This method avoids the instantiation of a fully featured Java object-tree to represent the XML-record and yields a smaller runtime memory-footprint.

We have run a series of tests with large numbers of messages, to figure out, what's the impact of our data-transition-model on the overall messaging performance. The results are displayed in section VI.

IV. Component Distribution Models

The described components give us the opportunity to model different distribution scenarios. Especially the deployment of message-based components allows a flexible infrastructure that helps minimizing the effects of network failures. A subtle layout of message relayers can also improve the performance of the entire network of accounting components, and it thus helps to deal with load issues, as we will show in this section. Furthermore the structure of our components enables the definition of different accounting domains that reflect the actual responsibility for the collected accounting data and the relationship between the participating resource providers and the administrative entities.

The main transport mechanism for messages in the accounting system is the message bus. We implement it by deploying a server instance of Apache ActiveMQ on every site that participates in routing and relaying. The resulting distributed message handling is a appropriate vehicle that can handle large message loads while complying with formal routing policies, that can be adjusted locally on every site, according to technical and legal requirements. Legal issues may arise when accounting data is indirectly submitted to the accounting authority, using third-party relays in the network. However, all message transfers in our accounting infrastructure are encrypted utilizing the cryptographic capabilities of the underlying transport technology. In case of the web-service binding that's the *WS-Security* specification, the message-channel uses the provided network connector security model, which is a configuration-item in ActiveMQ.

The basic component distribution is the classic client-server pattern, which deploys one central server, to which all the clients connect and submit their data. The resulting *single domain*, depicted in Figure 4 doesn't require relaying or routing, from an architectural point of view. Nevertheless, we can improve data safety and performance by applying the relay patterns that we discuss in next few paragraphs.

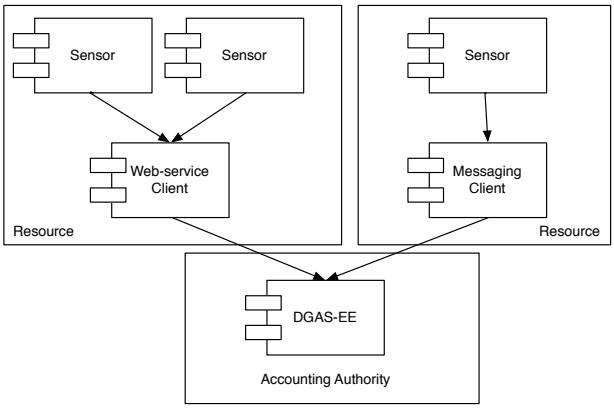

Fig. 4. Single Domain Case

A. Client-side Message Relay

One serious problem in a client-server based communication environment is the exception handling on the client side. A simple client that only wants to transmit uniform data records, has to be aware of a couple of difficulties that occur in case of a non-reachable server system. There are numerous strategies for a client-side incident response, but after a few retries and persisting data on a local storage, a client-component has to report the failure to the higher application layers, to take further action. The use of the messaging system simplifies this situation, because we can completely delegate the message handling to a self-contained subsystem with a decent failover-functionality and database-driven persistence. The fact, that the message broker and the client can be deployed on the same physical system minimizes the danger of network failures between client and broker. As a consequence, the client doesn't have to store the records locally for later transmission and it doesn't have to schedule the retries for the connection. The client-side relay approach suffers from the drawback, that we have to install a message broker, but on the other hand the client's exception handling can be kept simple. The client-side relay model enables us to tap infrastructures, where only a few sensors with high data-rates are operated, or where the sensor resides in a network segment that provides only limited reliability.

B. Site-centric Message Relay

In cases, where a number of accounting sensors are deployed, the client-side relay approach might cause a considerable overall-footprint. To achieve a more lightweight component distribution, we propose the site-relay model. In this scenario, we still keep the client-to-relay communication inside our own network-domain where we can control and maintain the connectivity. The site-centric relay acts as some sort of network concentrator, which routes the traffic over a dedicated connection factory with one single point of administration.

Store-and-forward Network Federation

By interconnecting the relay components in different domains, we construct a single overlay bus structure. In combination with the message addressing schema, all connected components are able to reach every participating component directly or via one or a number of relays. We provide different possible topologies for the connection of relays to the message bus, as sketched in Figure 5 and [9].

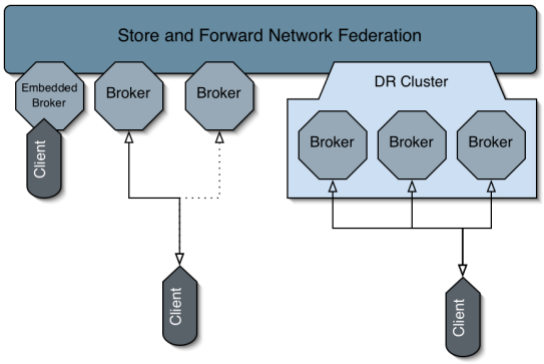

Fig. 5. Broker Topologies [9]

C. Multi-Domain Deployment

The deployment of the accounting system in a distributed multi-domain environment is the key feature of our concept. By applying the described components, we can model arbitrary component topologies. Figure 6 shows a possible case of multi-domain distribution.

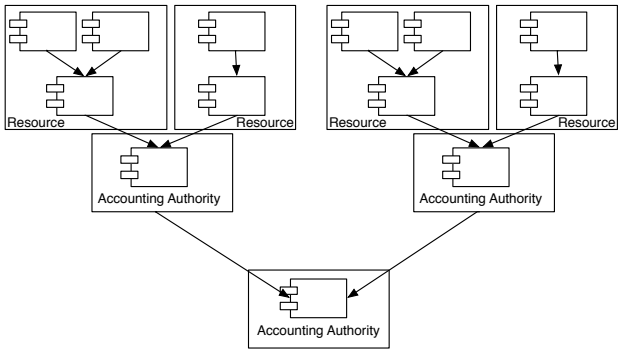

Fig. 6. Multi Domain Case

The convergence of the European Grids requires a hierarchical model of accounting authorities. Every participating European country runs its own *National Grid Initiative (NGI)* with

an appropriate facility to support accounting. The accounting of the *European Grid Initiative (EGI)* is the (vertically) higher-ranked authority that collects and correlates accounting data from the underlying national initiatives. In addition to such a scenario, we are able to define a number of horizontally arranged domains that only operate on a limited range of the accounting information. This would be useful to design a component distribution with separation-of-concerns, where an authority is responsible for selected excerpts of the accounting data.

The multi-domain deployment is different from the site-centric relay-model, because in the case of a simple relay, we do not deploy an entire accounting authority. Every domain in the multi-domain model has its own authority and a true responsibility for accounting data in the overall accounting-workflow.

V. Resource Discovery

In order to discover a resource for a particular job, the client or the meta-scheduler, respectively, issues an accounting record to the domain authority. This record is an estimated sample for the resulting job accounting record and contains the requirements for the job, but has no Job-ID. The case of the demo record in Listing 5, would be a request for a 32 CPU hardware. To make the request record more specific, additional elements may be provided, like a duration and a start-time proposal. If the runtime is unknown in advance, the duration element should be omitted. The accounting authority then returns the qualifiers of the computing elements that match the hardware requirements and that provide sufficient capacities. Accounting data of past jobs in similar configurations may be helpful for a runtime estimation [12], but in contrast to this proposal which focuses on capacity planning and management, our approach allows a rather short-term scheduling based on recent job submission data. However, even a rough estimation helps scheduling the jobs better.

VI. Implementation

Transparency and the integration of standard components is the fundamental requirement in the development of EEGAS. We have minimized individual programming, and even when doing so, we have tried not to cover the features of the underlying frameworks and subsystems, to make easy configuration possible. As an example, the configuration of the relay component is fairly transparent. The sample configuration in Listing 2 contains the URI's of the target message brokers. The parameter handling allows, that any valid broker URI-format, that is known to ActiveMQ, can be provided here. This includes failover and clustering capabilities, as well as transport options.

The prototype implementation of the accounting authority uses an automatically generated software layer that transforms the XML-records into DML-statements. This mechanism replaces the JPA-driven persistence in this early stage of the development. The software of the prototype can be downloaded at http://www.rrzn.uni-hannover.de/d-grid_accounting.html.

Performance Test

We have run a series of tests with large numbers of messages, to figure out, what's the impact of our data-format transition-model between the components (see Figure 3) on the overall messaging performance. The component setup for our test is displayed in Figure 7.

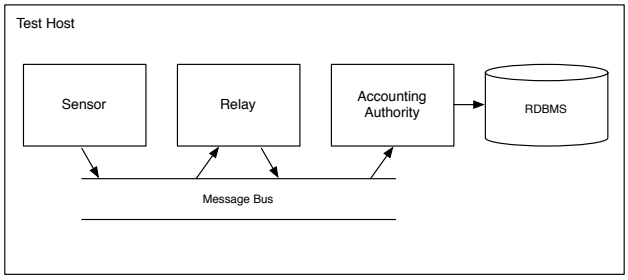

Fig. 7. Test Setup

The following list gives an overview on the software- and hardware-setup:

- 2x Intel Xeon 5150 @2.66GHz, 6GB RAM
- JBoss 5.1.0GA
- Apache ActiveMQ 5.2.0
- MySQL 5.1.6

We have chosen to install the entire setup on one physical host to be able to measure the application performance without the possible influences of the network. The message producer is the sensor component that generates CPU-accounting records like the *pending record* sample in Listing 5.

```
<UsageRecord xmlns="http://www.gridforum.org/2003/ur-wg">
  <RecordIdentity ur:recordId="urn:org:site:usage:82125.mach.org.site.ext:0"
    ur:createTime="2010-05-12T11:14:51.651+02:00"
    xmlns:ur="http://www.gridforum.org/2003/ur-wg"/>
  <JobIdentity>
    <GlobalJobId>82125.mach.org.site.ext</GlobalJobId>
  </JobIdentity>
  <UserIdentity>
    <LocalUserId>jdoe</LocalUserId>
  </UserIdentity>
  <Status ur:description="pbs_exit_status"
    xmlns:ur="http://www.gridforum.org/2003/ur-wg">0</Status>
  <Memory ur:metric="max" ur:storageUnit="KB" ur:type="virtual"
    xmlns:ur="http://www.gridforum.org/2003/ur-wg">1060991</Memory>
  <Processors ur:metric="total" xmlns:ur="http://www.gridforum.org/2003/ur-wg">
    32
  </Processors>
  <ProjectName ur:description="local_charge_group"
    xmlns:ur="http://www.gridforum.org/2003/ur-wg">
    g13563
  </ProjectName>
  <Host ur:primary="true" xmlns:ur="http://www.gridforum.org/2003/ur-wg">
    lomax.nas.nasa.gov
  </Host>
  <Queue>lomax</Queue>
  <StartTime>2010-05-12T11:14:51.651+02:00</StartTime>
  <Resource ur:description="pbs-jobname"
    xmlns:ur="http://www.gridforum.org/2003/ur-wg">
    m0.20a-7.0b0.0v
  </Resource>
</UsageRecord>
```

Listing 5. Sample Record

One test run consists of a series of 100.000 records that are passed to the message bus. The message bus features two message queues that are connected by the relay component in forwarder mode, so this part is simple. We need two configurations for the accounting authority to illustrate both cases of operation:

Setup	Messages	Time (sec)	Records per sec
A	100.000	2018	49,55
B	100.000	2165	46,19

TABLE I
PERFORMANCE FIGURES

- the AA performs direct persistence of the received accounting records (setup A)
- the AA translates the XML record into the Java object representation, replaces a value and passes the resulting XML record to the database (setup B)

These two test cases produce the figures in Table VI. The performance tests indicate a average performance of 48 records per second.

As we can also see, the construction of the Java object model and the serialization into XML text after the manipulation reduce the performance on the accounting authority by approximately 7% in the case of a record, as simple as the used sample. Larger records will cause higher load on the application server for the conversion between the data representations. Figure 8 outlines the number of records that have been collected by the accounting authority *DGAS2* of the current DGAS 3.4 installation at the Leibniz Universität Hannover in the first seven months of 2010. Compared to a theoretic throughput of our system of approximately 4M records a day, this performance seems feasible.

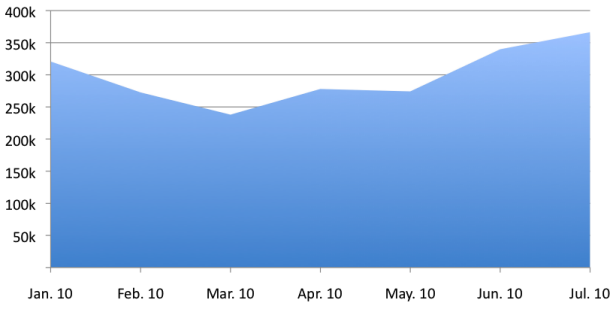

Fig. 8. D-Grid Records on *DGAS2* Jan-2010 to Jul 2010

The EEGAS system is currently being tested under realistic conditions in a parallel accounting infrastructure at the Leibniz Universität Hannover and the Universität Dortmund. Hannover uses the basic client distribution model (direct messaging) to account the records of the local grid cluster, whereas Dortmund have installed the site-relay model. In the test phase, EEGAS has processed the regular accounting records transmitted from both sites.

VII. CONCLUSION AND FUTURE WORK

This paper presents a distributed architecture based on Java-Enterprise components and message-driven communication. We have successfully implemented a prototype that can be used as a foundation to build an accounting network of arbitrary complexity, which reflects different responsibilities in an accounting workflow. We have implemented a Java model and a database schema that comprises the entire OGF usage record format. Our system is a good integration point for any accounting system that produces OGF records and can be used to implement capacity discovery as a foundation for meta-scheduling. We have performed a number of tests and have compared the resulting figures to the required throughput of an existing accounting authority in the D-Grid. The performance of the EEGAS prototype is sufficient to handle a multiple of the current load of a typical accounting authority.

In order to be able to provide a production-level software system, we will develop sensor components for new accounting metrics like software licenses or licensed data. We will integrate other accounting systems by providing mapping between different record formats and the OGF format. The performance of the software modules has to be improved to meet the requirements of a distributed, flexible and scalable accounting environment. To provide fine grained access to the recorded accounting data, we will develop interfaces to different visualization components, like HLRmon for DGAS [11].

ACKNOWLEDGMENT

Parts of this work are funded by the BMBF, the German Federal Ministry of Education and Research (PT-DLR grant 01IG07014).

REFERENCES

[1] A.Guarise, S.Bagnasco, R.Brunetti, A.Cristofori, S.Dal Pra, E. Fattibene, P.Veronesi, L.Gaido, G.Misurelli, G.Patania, P.Solagna, *Implementing a national grid accounting infrastructure with DGAS*, EGEE'09, September 2009, Barcelona, Spain.

[2] R.M.Piro, M.Pace, A.Ghiselli, A.Guarise, E.Luppi, G.Patania, L.Tomassetti, A.Werbrouck *Tracing resource usage over Heterogeneous grid platforms: A prototype RUS interface for DGAS*, E-Science 2007, December 10-13 2007, Bangalore, India.

[3] A. Guarise *Accounting - toward national grid infrastructures*, HPDC workshop on Monitoring, Logging and Accounting, (MLA) in production Grids, 2009, Munich, Germany.

[4] R. Byrom et. al., *APEL: An implementation of Grid accounting using R-GMA*, UK e-Science All Hands Conference, Nottingham, September 2005.

[5] M. Jiang, C. Del Cano Novales, G. Mathieu et al., *APEL: A CPU Accounting Infrastructure for Grids*, ISGC2010, March 2010, Taipei, Taiwan

[6] R. Mach, R. Lepro-Metz, S. Jackson, *Usage Record - Format Recommendation*, The Global Grid Forum, 2003.

[7] S. Crouch, D. Fellows, X. Chen, *Experiences of Using Usage Record (UR) version 1.0*, The Open Grid Forum, 2009.

[8] The JBoss Community, *JBoss AS Community Edition* http://www.jboss.org.

[9] The Apache Software Foundation, *Apache ActiveMQ* http://activemq.apache.org/topologies.html.

[10] B. Coghlan, A. W. Cooke, A. Datta et al., *R-GMA: A GRID INFORMATION AND MONITORING SYSTEM*, WP3, UK e-Science all hands conference, Sheffield, September 2002

[11] S. Dal Pra, E. Fattibene, G. Misurelli et al., *HLRmon: a Role-based Grid Accounting Report Web Tool*, CHEP'07, September 2007, Victoria, Canada

[12] R.M. Piro, A. Guarise, G. Patania and A. Werbrouck, *Using historical accounting information to predict the resource usage of grid jobs*, Elsevier Science Publishers B. V. 2009, Future Gener. Comput. Syst. Journal Vol.25 No.5, DOI http://dx.doi.org/10.1016/j.future.2008.11.003