Heimdhal: A History-based Policy Engine for Grids

Pedro Gama, Carlos Ribeiro, Paulo Ferreira
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol, nº9, 1000-029 Lisboa, Portugal
[pedro.gama, carlos.ribeiro, paulo.ferreira]@inesc-id.pt

***Abstract*— The arising of grid platforms introduced inexpensive and highly available computing, storage and networking resources. Therefore, in a worldwide trend, institutions aggregate on virtual organizations, registering their resources to the grid and in return accessing a virtually limitless warehouse. This overabundance allowed the emergence of innovative application and business models, delivering the solution to several large-scale problems, as is the case of data processing, storage and sharing in CERN's Large Hadron Collider Project.**

In order to allow system administrators to assure resources are employed in a coordinated and secure way, policy mechanisms need to cope with such new models and the increased complexity in resource usage management. However, current grid platforms only provide simple primitives in their authorization modules. By restricting access control mechanisms to ACLs and role-based models, they disregard powerful usage semantics, such as those which are history-based (e.g. the chinese-wall security policy). This absence obliges the development of ad-hoc security managers for each deployed resource, introducing vulnerabilities in the security architecture.

The use of advanced policies, and more specifically history-based policies, provides a natural method for expressing and enforcing several grid usage patterns, such as fair resource consumption. Additionally, some policy concepts not usually found in policy engines, such as periodic reevaluation, assure an effective policy enforcement.

We present the Heimdall system, a history-enabled policy engine which allows the definition, enforcement and accounting of history-based policies in Grid platforms, and more specifically in Globus Toolkit 4.0. A practical evaluation using selected usage patterns corroborates the effectiveness of this kind of policies in grid computing environments, denoting encouraging performance results.

I. Introduction

Several usage constraints must be applied in runtime systems in order to control access to system resources. One approach for the definition of such constraints is based on the use of security policies[1]. These policies describe the access rules with an high level of abstraction, clearly separating the specification and the implementation of security mechanisms[2]. In particular, history-based policies allow the inclusion of temporal events in the policy rules. A common example of such type of policies is the Chinese-Wall security policy[3], which analyzes previous actions in order to authorize (or not) the current operation.

Additionally, the specification of usage rules in a high-level language aids policy administrators in generalizing usage and security patterns into relevant policies. These policy patterns can be stored into policy repositories, for later instantiation in similar situations. This model reduces the effort associated with policy administrative tasks and significantly increases response time to environmental and business changes.

Specifically, in grid environments, we can envision several usage and security patterns for which history-based semantics apply:

1) A Scheduler wants to assure a minimum CPU QoS to certain submitted jobs. Thus, it rejects job executions in nodes where CPU-intensive applications are already running.
2) A Scheduler wants to impose a tit-for-tat model[4] in resource provisioning. Grid users start with some amount of pre-defined resource credits, but must share their resources in order to obtain further credits.
3) A system administrator wants to prevent resource consumption associated with erroneous software. He specifies that any software that has terminated abruptly in the past cannot execute again before being validated (thus obliging the application owner to become aware of the problem).
4) A network-service provider wants to limit resource consumption to a certain amount per week (e.g. each user has a 10GB usage capacity each week). Additionally he wants to reserve 10% of the capacity for its local users.
5) A CPU-service provider wants to enforce usage limits to jobs submitted by users. Not only must the policy engine deny new submissions when the user has exceeded his quota, but also jobs must be cancelled if they exceed those limits while executing.
6) A resource manager wants to restrict job submissions in situations where requests are continually rejected (e.g. due to resource exhaustion). Therefore it automatically rejects any job from a user who had more than 2 job submissions rejected in the past 5 minutes.

These usage scenarios are obviously very dynamic, and the policy administrator needs to be able to modify them quickly according to business conditions. For instance, in the course of a marketing campaign, a network service provider might decide to add a new rule to its policy base, granting unlimited access during night hours. Of course this requirement can be implemented in an ad-hoc manner. Effectively, the lack of flexibility and expressiveness in existing grid policy engines obliges policy administrators to code policy rules into the resource managers themselves. However, this increases

time-to-enforcement regarding new policies, while reducing application portability and contributing to vulnerabilities in the policy architecture.

In today's business reality, this model simply isn't acceptable. Policy Administrators must be able to define and modify policies in a user-friendly manner, and subsequently enforce them immediately upon its resources. We believe that, by providing a richer policy semantics, administrators will be drawn into using an higher-level policy definition language, demarcating themselves from application-level resource manager customization.

The concept of history-based policies has already been incorporated in several policy engines[5], [6], [7]. However, to our knowledge, only the SESAME system[8] proposes to integrate this kind of policies in grid platforms. Their architecture for policy enforcement is based in finite-state machines, which can show serious scalability problems when the number of policies and/or complexity increases. In addition, the SESAME platform doesn't incorporate reevaluation semantics (c.f. section II).

The platform presented in this paper, called Heimdall, allows the specification and enforcement of several policy semantics in grid platforms, and more specifically in Globus Toolkit 4.0. Policy administrators can define history-based policies independently of application development and subsequently generate a PDP (Policy Decision Point) in order to support its enforcement. In addition, Heimdall provides extensible ready-to-use policy patterns for established usage scenarios, such as limited resource sharing, fair resource sharing, scientific data cooperative analysis, etc. To our knowledge this is the first practical design, implementation and evaluation of a platform that supports the specification and enforcement of history-based usage and security policies in grid environments.

This paper is organized as follows. In the next section we present a general overview of Heimdall, describing a generic application execution. In Section III we define our policy language: the xSPL. We then present the architecture of the system in Section IV, describing the various components of the platform. In Section V, we focus on the prototype implementation, further evaluated in Section VI. Then, we contextualize Heimdall within this scientific field discussing some related work in Section VII. Finally, in Section VIII we present our conclusions.

II. System Overview

Heimdall is a Middleware platform with the architecture depicted in Figure 1, which was derived from the following design goals:

1) Provide an expressive, yet user-friendly, language for history-based policy definition.
2) Implement a modular architecture that allows an easy integration with current grid platforms, as well as with grid standards and specifications being currently defined.
3) Clearly separate policy specification and enforcement from application development.

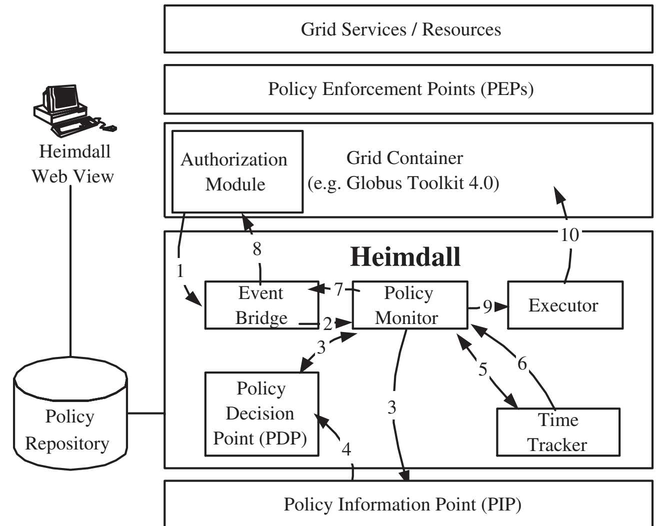

Fig. 1. Overview of Heimdall.

4) Generalize grid usage and security patterns into relevant policies, and allow their instantiation and deployment through a web interface.

In what concerns the first goal, Heimdall extends the SPL (Security Policy Language)[5] in order to increase its expressiveness for grid policy enforcement. As described in the next section, the resulting policy language, xSPL, allows the specification and enforcement of a large number of grid usage and security patterns, which are not supported by current policy engines.

Seamless integration with several grid service containers (e.g. the Globus Toolkit 4.0) is provided through Heimdall interface modules, more specifically the Event Bridge and the Executor. The input to Heimdall is generated in the container authorization modules by intercepting security critical operations (e.g. access to a grid resource). On the other hand, Heimdall contacts the grid container for the invocation of policy-dependent custom actions (e.g. terminate a job that has exceeded its CPU usage quota) through standard protocols (e.g. invoking a webservice). This architecture allows any grid container to use our policy engine with minor integration efforts. Additionally, Heimdall may be used in parallel with other policy and security engines without any interference.

Separation of application development and policy specification and enforcement is also a key target in Heimdall. The grid container only has to make sure that a notification of every relevant operation (called an event) is generated upon execution. The policy administrator will then define the history-based policies based on these event descriptions. This approach allows the integration of events from different applications into a common policy without the need for additional application development.

Heimdall Web View is a web interface that significantly reduces the administrative effort associated with policy administration. Not only it allows policy administrators to visualize and edit current deployed policies, but also allows them to add

new policies to the system through the instantiation of predefined and extensible usage and security patterns (e.g. define temporal limits for resource consumption). Finally, Heimdall Web View allows the accounting of deployed policies (e.g. was this policy activated by user 'John Doe' during last week?).

For clarity we illustrate in Figure 1 the access to a generic resource over a grid container using Heimdall mechanisms to enforce history-based policies.

1) The Container Authorization Module sends an operation description to Heimdall.
2) The operation description is received in Heimdall by the Event Bridge Module. This module translates the operation description into a normalized construct: the xSPL event. This event is then sent to the Policy Monitor for overall policy coordination.
3) The Policy Monitor contacts the Policy Decision Point (PDP) in order to determine if the current event is applicable for any of the deployed policies. In the affirmative case, the event must be stored in the Policy Information Point (PIP) for future evaluations.
4) The PDP retrieves applicable past events from the PIP in order to analyze the evolution of the policy.
5) Additionally, if the current event effects cannot be determined at current time (e.g. job execution period is not known at submission time), the Time Tracker creates a new timer for periodic policy reevaluation. On the other hand, when the policy termination condition is fulfilled (e.g. a job concludes or fails), the Time Tracker is contacted to cancel an existing timer.
6) At periodic intervals (as specified in policy definition), the Time Tracker contacts the Policy Monitor in order to reevaluate the policy, and thus check for policy infractions during the action execution.
7) After overall evaluation is concluded, the Policy Monitor returns the authorization decision (i.e. a Authorize or Deny statement) concerning the execution of the specified action.
8) The Event Bridge translates the Authorization decision into the Grid Container format and returns the security information to the Authorization Module in the Container.
9) If the Policy Monitor verifies an infraction in a policy executing instance (e.g. caused by a running job), it invokes the Executor in order to execute the policy-defined custom action.
10) The Executor serves as the interface between Heimdall and the Grid Container in what concerns the execution of policy-dependent custom actions (e.g. terminating or reducing the priority of a certain job). This can be achieved using a number of protocols (e.g. webservices, HTTP, RMI, SMTP, etc), providing a flexible connector to Grid Resource Managers.

III. xSPL

Heimdall policies are defined in xSPL (eXtended Security Policy Language), an extension of the SPL language[5]. This language allows the definition of policies with complex constraints, including history-based policies.

A. xSPL Basic Constructs

Each policy is defined by the composition of several rules. Each rule is formed by two distinct sections:

$$< trigger\ expression :: decision\ expression >$$

The trigger expression specifies the applicability of the rule. If this expression is true, it means the decision expression part must be evaluated, and its result will constitute the authorization decision for the present rule.

One of the central concepts in xSPL is the event. Each action in the system, like a job submission or a resource consumption, is mapped into a normalized event construct. An event can either be instantiated in the Past Events Set (denoted as 'pe' in Figure 2), which represents the events previously executed, or it can represent the operation being evaluated, called the current event (always denoted as 'ce' in Heimdall).

For clarity, we present in Figure 2 a definition in xSPL of a policy intended to promote resource usage by large CPU-intensive jobs (indicated by an execution period exceeding 15 minutes).

```
JobsSubmitted = PastEvents @{.action = "Submit_Job"};
JobsConcluded = PastEvents @{.action = "Job_Concluded"};
JobsNotConcluded = JobsSubmitted @{
  .action.parameter[0] NOT IN
  JobsConcluded.action.parameter[0]};

policy PromoteLargeJobsPolicy {

?PromoteLargeJobs():
  EXISTS AT MOST 3 pe IN JobsNotConcluded
    ce.action = "Submit_job"
    ::
    ce.time - pe.time > 15 minutes
}
```

Fig. 2. Example of a history-based policy in xSPL (note that the parameter[0] of the "Submit Job" and "Job Concluded" events contains the job id).

This policy states that a job submission action (the current event) is authorized only if no more than 3 jobs are executing for 15 minutes. Otherwise, job submissions are denied. Although the task of defining this rule is several orders of magnitude easier than coding the rule into the authorization platform, we additionally allow policy administrators to instantiate policy patterns in Heimdall web interface (named Heimdall Web View).

B. xSPL Reevaluation Mechanisms

A significant extension to Heimdall (w.r.t. SPL expressiveness) relating history-based policies is the possibility to define policy reevaluation properties (e.g. after a job submission, and while it is running, verify each minute that usage quota is not exceeded). Existing history-based policy engines[5], [6], [7] issue an authorization decision based solely in the evaluation of the policy rules at the moment an action is requested. However, we find this model to be restrictive for a number of scenarios in grid environments.

Imagine for example the policy specified in Figure 3.

```
JobsSubmitted = PastEvents @{
        .user = ce.user AND .action = "Submit_Job"};
JobsSubmittedLast30Days = JobsSubmitted @{
        .time - ce.time < 30 DAY};
JobsConcluded = PastEvents @{
        .user = ce.user AND .action = "Job_Concluded"};
JobsConcludedLast30Days = JobsConcluded @{
        .time - ce.time < 30 DAY};
JobsContainedLast30Days = JobsConcludedLast30Days @{
        .action.parameter[0] IN
        JobsSubmittedLast30Days.action.parameter[0]}
JobsNotContainedLast30Days = JobsConcludedLast30Days @{
        .action.parameter[0] NOT IN
        JobsSubmittedLast30Days.action.parameter[0]}
JobsNotConcluded = JobSubmitted@{
        .action.parameter[0] NOT IN
        JobsConcluded.action.parameter[0]}
TotalTimeJobsConcludedInLast30Days =
        JobsContainedLast30Days.SUM(.action.parameter[1])
        + JobsNotContainedLast30Days.SUM(
           .action.parameter[1] - (ce.time - 30 DAY));
TotalTimeJobsNotConcluded = JobsNotConcluded.SUM(ce.time -
        .time);

policy RestrictedCPUUsagePolicy {

?RestrictCPUUsage ():
  ce.action = "Submit_job"
  ::
  TotalTimeJobsConcludedInLast30Days +
    TotalTimeJobsNotConcluded < 10 HOUR;
}
```

Fig. 3. Example of a CPU Usage Restriction policy in xSPL (note that the parameter[0] of the actions contains the Job ID and the parameter[1] of "Job Concluded" event contains the job execution time).

This policy restricts the user CPU usage to 10 hours in a 30-days period. In a traditional policy model, the policy engine is only enabled at job submission time, in order to validate if the user has consumed less than 10 hours of CPU during the current month. This semantics implicates a deficient enforcement of history-based policies, in cases where the overall effect of the authorized actions is not known at evaluation time. That is the case in the above policy, where the job, after preliminary authorization, could continue to execute way beyond the user 10h CPU quota.

We agree that, in certain environments, the policy engine can optimistically authorize the actions, supported by the assumption that it can impose compensating penalizations in the future to any infringing user. Effectively, this model of operation is even supported by Heimdall, through the definition of an obligation policy[9].

However, in dynamic grid scenarios this optimistic approach is rather difficult to implement in practice. The high dynamics (even volatility) associated by nature to these environments, imply that a certain user (or even an entire organization) may use a certain grid node resource only once in a lifetime. Furthermore, it is reasonable to assume that a number of identical resources (such as CPU) are available from different sources in the virtual organization.

These facts make it extremely difficult to impose penalties towards a certain user, as he can simply start using other nodes in the network (or even other virtual organizations if a VO-wide policy is enabled).

In order to enforce a pessimistic, yet safer and more effective approach, Heimdall extends traditional history-based policy semantics, in order to impose a periodic reevaluation of relevant policies. This allows the effective enforcement of the policy, whilst not obliging resource manager intervention (e.g. by coding additional resource usage restrictions) and without noticeable performance penalties (reevaluation period is custom defined for each policy, thus eliminating unnecessary evaluations).

More specifically, Heimdall allows the policy administrator to enhance the policy specification with the following reevaluation primitives:

- Reevaluation Trigger Condition
  Defines the condition on which Heimdall starts a new timer to schedule policy reevaluation (e.g. the current event states a "Job Submission").
- Reevaluation Concluded Condition
  Defines the condition on which Heimdall stops a current reevaluation timer (e.g. the current event states a "Job Concluded").
- Reevaluation Match Parameter
  Defines the parameter that should match in the current event of the trigger and concluded conditions in order for a timer to be canceled (e.g. a parameter in the "Job Submission" and "Job Concluded" events, with the job ID).
- Reevaluation Periodicity
  Defines the periodicity on which Heimdall should reevaluate a policy. By default Heimdall reevaluates the policy once each minute.
- Infraction Stop Action
  Defines the action that Heimdall should invoke when an infraction of the policy is detected (e.g. stop an infringing job).

In the case of the policy defined above in Figure 3, the reevaluation tags might be defined as follows:

```
REEVAL_TRIGGER      : ce.action = "Submit_job";
REEVAL_CONCLUSION   : ce.action = "Job_Concluded";
REEVAL_MATCH        : ce.parameter[0];
REEVAL_PERIOD       : 15 MINUTE;
REEVAL_INFRACTION   : "http://myGridContainer/JobScheduler?
action=cancelJob&id=" + ce.parameter[0]"
```

Fig. 4. Reevaluation tags for the CPU Usage Restriction policy.

IV. Architecture

Heimdall possesses a modularized architecture based in several independent modules, as depicted in Figure 1.

The Event Bridge receives action descriptions from the authorization modules of Grid Containers, and creates a normalized description (referred to as a xSPL event). In a common Heimdall deployment, several Event Bridges can coexist, connected with several authorization modules, denoting different action descriptions and/or interfaces. It is interesting to notice that in some situations the Event Bridge can receive events

from the Heimdall engine itself, derived from policy enforcement actions (as is usual in common obligation policies).

The Executor module is necessary in order to assure that, in case of policy infraction, system consistency is restored in a predefined manner. This module is contacted by the Policy Engine when a policy instance is reevaluated and an infraction is detected. It then executes the actions specified in the policy definition for these situations (which usually consist in terminating an execution by contacting the Grid Container or the Resource Policy Enforcement Point itself). It is important to point out that in policy engines that don't support periodic reevaluation, there's no need for an Executor-like module. In that case policies are either authorized or denied upon action submission, and the effects of the action are presumed to take place immediately, thus preventing policy infractions. Obviously, as stated in the previous section, that is not usually the case in practical situations, like those stated in the usage scenarios, and thus the need for reevaluation mechanisms in the policy engine.

The Time Tracker provides time-related mechanisms to control the reevaluation of policies.

The Policy Monitor and the Policy Decision Point (PDP), are the core of the Heimdall platform. They are described below in more detail.

The Policy Information Point (PIP) is the event repository in Heimdall. It filters and stores the events which are relevant for policy evaluation.

Finally, while the Policy Repository holds all the policy specifications, the Heimdall Web View allows a policy administrator to create and edit policies in a user-friendly manner. In addition, he can also instantiate and/or extend policy patterns, thus enforcing common usage scenarios with minimal effort and increased quality assurance.

A. Policy Monitor

The Policy Monitor module is the overall coordinator of Heimdall. It manages policy decision queries received from the Grid Container authorization modules (through the Event Bridge). These queries are forwarded to Heimdall Policy Decision Point (PDP), which returns a DENY/AUTHORIZE/NOT APPLY decision. In addition, it assures the system is in a consistent state with respect to history-based policies, by enforcing policy reevaluation whenever applicable, and activating custom-defined actions when an infraction is detected. For this task, the Policy Monitor manages policy instances whenever policy reevalutions are necessary, by storing the policy ID, a reevalution timer and the current event (in order to assure a similar reevalution of the policy).

In order to illustrate these concepts we represent in Figure 5 a generic execution of a system enforcing the RestrictCPU-Usage policy presented previously in Figure 3.

Initially events A and B are received by Heimdall, representing job submission requests belonging to the same user. They are processed by the Policy Monitor, which uses the PDP in order to assess authorization decisions. The RestrictCPUUsage rule applies to these events, and both decisions authorize the

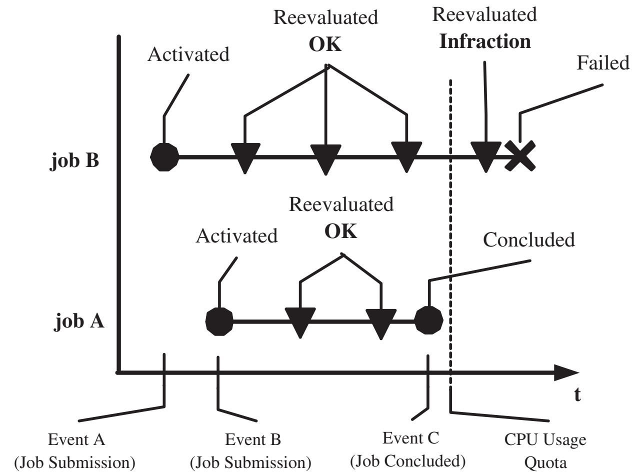

Fig. 5. The evolution of two job executions

job. However, as this policy requires a periodic reevaluation, the Policy Monitor requests a timer for each of the policy instances from the Time Tracker Module.

Afterwards, at periodic intervals, policy instances associated with events A and B are independently reevaluated, in order to validate that the system remains in a consistent state (i.e. that the User CPU Usage Quota wasn't exceeded).

After some time, event C is received, denoting a Job Conclusion that matches event B Job Submission. This fact is verified by the PDP, at Policy Monitor request. The Policy Monitor thus contacts the Time Tracker to cancel the associated timer.

At a later instant, the CPU Usage Quota is effectively exceeded, a condition that is verified by the PDP through the DENY decision in the evaluation of the RestrictCPUUsage policy. This fact directs the Policy Monitor to terminate all associated resource/service usage (in this case the job submission generated by event A). Additionally to canceling the timer in the Time Tracker Module, the Policy Monitor commands the Executor to terminate the job by contacting the Grid Container.

B. Policy Decision Point (PDP)

The mechanism associated with the Policy Decision Point (PDP) basically evaluates a certain policy against the current event and the content of the PIP. Following evaluation, the mechanism returns one of three possible results (for clarity we consider the RestrictCPUUsage policy to illustrate PDP operation):

- It returns ALLOW if the trigger and obliged expressions are true (i.e. the current event denotes a Job Submission and the User CPU Usage Quota wasn't exceeded).
- It returns DENY if only the trigger expression is true (i.e. the current event denotes a Job Submission and the User CPU Usage Quota is already exceeded).
- It returns NOT-APPLY if the trigger expression is false (i.e. the current event doesn't denote a Job Submission).

It is important to notice that the policy administrator can either define policies using a DENY-overrides (a DENY decision in one of the rules finalizes the evaluation) or a AUTHORIZE-overrides (an AUTHORIZE decision in one of the rules finalizes the evaluation), as referred in [10].

V. Implementation

In order to increase code portability, Heimdall was implemented in Java, and deployed using Java Runtime Engine 1.4.2. The current prototype implementation is integrated with Globus Toolkit 4.0 (GT4) authorization modules, as described below. Work is in progress towards integration with other Grid engines, namely Condor.

For lack of space, we focus this section on Heimdall mechanisms for job execution control in GT4. However, extrapolation to other kinds of resource usage control should be easy.

In the first place, the Event Bridge must receive a description of relevant actions that are requested from the Globus Container, in order to authorize/deny its execution. Thus, we extended GT4 ManagedJobService classes in order to contact Heimdall Event Bridge and generate "Job Submission", "Job Concluded" and "Job Failed" action descriptions. We consider this to be a simple and efficient approach, yet allowing us to implement all job-related policies already mentioned.

The Policy Decision Point is an extension of the previous Heimdall Evaluation Engine implementation, itself extending the reasoning engine proposed by Ribeiro[5] with additional history-based and obligation semantics.

The Policy Monitor controls the overall deployment and enforcement of policies in the system. For that purpose it holds a vector with the identifier of all deployed policies in the system, and another vector with a descriptor for all history-based policy reevaluation instances (i.e. policies involving an action that is still executing). This latter vector holds, besides the policy identifier, the current event that was authorized at submission time, and a timer object, received from the Time Tracker Module. Whenever an event is received from the Event Bridge, the Policy Monitor uses the Policy Decision Point to evaluate the event against all deployed policies. Additionally, it also checks if the event concludes any of the executing history-based policy instances. In what concerns the execution of custom actions upon policy infraction, the Policy Monitor possesses a distinct thread of execution, which is activated by the Time Tracker module in case a timer expires.

The Policy Information Point is implemented as a set of event repositories, one for each of the policies in the system. At policy deployment time, each of these repositories is dynamically generated according to policy definitions, in order to optimize event storage and retrieval. A filtering mechanism discards any event that is not relevant for the evaluation of the policy (e.g. if a policy controls a payment service, it doesn't need to store events related to file access). In addition, the repository stores only useful evaluation fields (e.g. if a policy controls the number of books sold, it doesn't have to store the price of the books). This approach allows us to minimize Log dimension, and thus enhance performance of event retrieval (for more details see [5]).

Finally, the Executor Module parses and invokes the defined custom actions. More specifically, in the case of job termination in GT4, the Executor contacts the GT4 Job Management Web Service in order to terminate the specified job. Additionally, HTTP-based callbacks (e.g. *http://myserver/customAction.jsp?id=1*) are available, and new interface modes are being developed and will be available shortly.

VI. Evaluation

In order for Heimdall to be of practical use, a grid container should be able to request an authorization decision without incurring in significant performance penalties.

As is usual in history-based policy engines, Heimdall performance is directly related to the time it takes to search past events in the Policy Information Point (policy evaluation time increases with the number of events in the PIP).

Furthermore, Heimdall's reevaluation semantics oblige the management of policy reevaluation instances. On each event reception, the reevaluation trigger and conclusion conditions must be verified, in order to potentially start a new policy instance or finish an existing one. It is important to notice that, in the context of the reevaluation concept, only the instance management (i.e. create and finish instances) has influence in the system performance. The policy reevaluations themselves have only minor effects, as their periodicity can be customized according to the system characteristics.

In order to assess the adequacy of Heimdall in what respects performance, we performed a practical simulation in a Pentium 4, 2.8GHz, 512MB PC, running Linux Suse 9.3.

Consider the following usage scenario, presented in Figure 6.

```
providedGridJobs = PastEvents @{
     .author.homeHost != ce.target.homeHost &
     .target.homeHost = ce.target.homeHost &
     .action.name = "Job Concluded"
  }
consumedGridJobs = PastEvents @{
     .author.homeHost = ce.target.homeHost &
     .target.homeHost != ce.target.homeHost &
     .action.name = "Job Concluded"
  }

policy FairUsage{
REEVAL_TRIGGER      : ce.action = "Submit job";
REEVAL_CONCLUSION : ce.action = "Job Concluded";
REEVAL_MATCH      : ce.parameter[0];
REEVAL_PERIOD       : 15 MINUTE;
...
?FairUsage:
  ce.action.name = "Job Submission"
  ::
  providedGridJobs.SUM(.action.parameter[1].value) >
      consumedGridJobs.SUM(.action.parameter[1].value)
}
```

Fig. 6. Example of a CPU Fair Usage policy in xSPL (note that the parameter[1] of "Job Concluded" event contains the job execution time).

The FairUsage policy specifies that an organization cannot consume more CPU resources than it has provided in the past.

This is a common grid usage scenario, which can be deployed in a regular virtual organization scheduler.

We assume a virtual organization with 10 hosts (an host can represent more than one container) and 50 users per host. In addition, the scheduler never selects the user's home host for execution, and the usage is always authorized (i.e. each host has provided more than it has consumed in the past).

For lack of space, we restrict the presented evaluation to a single policy. However, further evaluations show that individual policy reevaluation time is independent of the number of policies in the system (e.g. evaluating two events for the same policy, or the same event for two similar policies takes the same time).

During the simulation we iteratively submit a job for each one of the users (500 in total), and later conclude the jobs. Thus the average amount of reevalution instances in the system is 250 (half the number of submitted jobs). Additionally, at simulation setup time, we submit a number of auxiliary jobs (0, 500, 1000, 2000 and 4000 respectively for each of the presented runs), which are never concluded, thus creating a number of pending reevaluation instances. This allows us to test the effects of the reevaluation concept in the overall policy evaluation time.

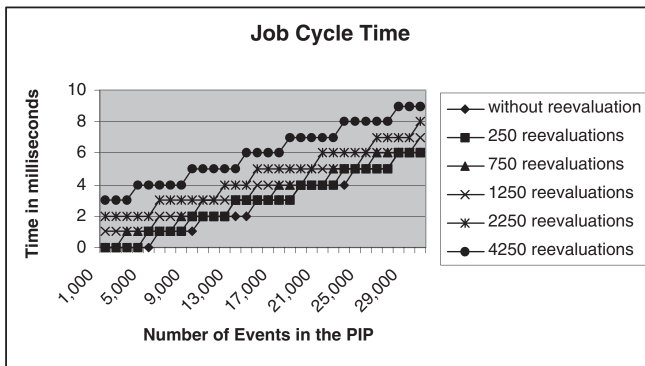

Fig. 7. Job Cycle Evaluation Time in the FairUsage Policy.

The lines in the chart presented in Figure 7 correspond to different simulations with a distinct number of reevaluation instances in the system (from 250 to 4250 in average). The values in the chart were obtained by running job cycles (job submission plus job conclusion) and averaging the evaluation time every 1,000 cycles. Notice that each average is presented in the end of the associated interval (e.g. the average evaluation time in the 10,000-11,000 interval is presented in the 11,000 index) and every job cycle corresponds to two events in the PIP.

The obtained results indicate a rather acceptable performance for the FairUsage Policy. The job cycle time ranges from nearly 0 to 9 milliseconds, a clearly negligible result when compared with the 403 milliseconds average job submission time we obtained with Globus Toolkit 4.0 (Heimdall policy evaluation takes about 1% of total job submission time).

Furthermore, the results show that the system scales efficiently with the increase in the number of events in the Policy Information Point and the number of policy reevaluation instances.

VII. Related Work

Due to the fact that existing grid platforms only provide limited expressiveness in authorization modules, a number of policy engines was integrated with these platforms. Each one addresses a specific issue, but they all lack Heimdall's expressiveness and enforcement capabilities with respect to history-based policies.

CAS (Community Authorization Service)[11] is the policy engine that is bundled with Globus Toolkit[12]. It allows local grid administrators to delegate the definition of users and privileges to a central authority in the virtual organization. Alternatively, these definitions would need to be replicated throughout the virtual organization. However, local resource managers still have to be modified in order to enforce the defined policies. Furthermore, CAS policies are based in a RBAC model, which by itself lacks expressiveness to enforce the usage scenarios we presented in Section I. It's interesting to note that Heimdall itself can be deployed locally in each resource manager to interpret and enforce CAS-defined policies.

Akenti[13] is an authorization platform which issues assertions containing the user privileges in what concerns a certain resource. Policies and assertions are specified in a custom XML policy language[14] and, as is the case with CAS, Akenti is not concerned with policy enforcement, which it delegates to the resource manager[14].

Cardea[15] was developed as part of NASA Information Power Grid to manage authorization requests. It is based upon the XACML standard for policy specification and SAML for assertion communication between platform components. However it doesn't support history-based policies.

PRIMA[16] focus on access control for small and dynamic working groups. Thus, besides allowing the management of resources privileges using an ACL and RBAC approach, it also enforces these privileges through the dynamic configuration of local user accounts. It uses X.509 Attribute Certificates for policy specification and is currently integrated with POSIX file system ACLs and the Grid Access Control Lists (GACLs) of project Slashgrid. However, as is the case with PERMIS[17], which is also based in X.509 Attribute Certificates, PRIMA lacks the support for history-based policies.

Sundaram[18], [19] describes a policy framework for usage management in grids. Policy rules are defined in XML files using a custom language based in attribute-value pairs. It allows policy administrators to impose resource consumption limits and additionally automates task submission privilege validation on the user side.

Keahey et al[20], [21] integrated a policy engine into the Globus GRAM (Grid Resource Allocation and Management) module. It allows the definition of fine-grain usage rules using an extension to RSL (Resource Specification Language), that contains tags such as ”action”, specifying the semantics associated with the job. The presented tags are hardly suitable

for complex usage scenarios, being focused for simple job management within the Virtual Organization.

Verma et al[22] proposed a policy service for grid computing which intends to provide cooperating VO-level and local policy definition and enforcement. However, history-based policy support is not mentioned.

The SESAME[8] proposes a DRBAC (Dynamic Role-based Access Control) model for grid authorization control. DRBAC is implemented by incorporating a state-machine that allows the evaluation of environment variables (e.g. current time and location) in addition to traditional RBAC semantics. History-based policies can be defined and enforced by using these state-machines. However, scalability issues are paramount with this approach, and the small-scale examples shown in the presented evaluation fail to prove SESAME's usefulness in real world scenarios, containing a large number of nodes in the state-machine.

Wasson[23] proposes a system to support resource provisioning in virtual organizations (VOs)[12]. The system is centered around the concept of a Grid Bank which provides users with credits to use the VO'resources. Although including different components for policy monitoring, accounting and enforcement, it is focused on controlling just three types of policies involving provisioning, which limits its applicability.

In conclusion, all these systems lack the necessary semantics for effective history-based policy enforcement.

VIII. Conclusions

In this paper we evidenced the relevance of history-based policies in the context of grid platforms. We showed several usage scenarios, in which the required usage semantics cannot be defined using traditional authorization mechanisms, thus obliging ad-hoc extensions to the security architecture. Furthermore we demonstrated the adequacy of the policy reevaluation concept for history-based policy enforcement.

We developed a prototype of the Heimdall platform, which allows the specification and enforcement of history-based policies in Globus Toolkit 4.0. In addition, this platform can be easily integrated with other grid containers through its interface modules.

Finally, we found the performance penalization associated with Heimdall operation to be negligible. Our evaluations show that the performance of the Heimdall platform in terms of event processing is adequate for practical situations, efficiently scaling for a large number of events in the Policy Information Point and thousands of policy reevaluation instances.

References

[1] J. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the IEEE Symp. Security and Privacy*, California, USA, 1982.

[2] T. Y. C. WOO and S. S. Lam, "Authorizations in distributed systems: A new approach," *Journal of Computer Security*, vol. 2, no. 2,3, pp. 107–136, 1993.

[3] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, 1989, pp. 206–214.

[4] B. Cohen, "Incentives build robustness in bittorrent," in *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, USA., June 2003.

[5] C. N. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes, "SPL: An access control language for security policies with complex constraints," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, Feb 2001.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proceedings of the 2nd International Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, 2001.

[7] J. Lobo, R. Bhatia, and S. Naqvi, "A policy description language," in *Proceedings of the National Conference of the American Association for Artificial Intelligence*, Florida, USA, 1999.

[8] G. Zhang and M. Parashar, "Dynamic context-aware access control for grid applications," in *Proceedings of the Fourth International Workshop on Grid Computing (GRID 03)*, 2003.

[9] P. Gama and P. Ferreira, "Obligation policies: An enforcement platform," in *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, Stockholm, Sweden, June 2005.

[10] T. Freeman and R. Ananthakrishnan, "Authorization processing for globus toolkit java web services," *IBM developerWorks*, October 2005. [Online]. Available: http://www-128.ibm.com/developerworks/grid/library/gr-gt4auth/

[11] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke, "A community authorization service for group collaboration," in *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, 2002, p. 50.

[12] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," in *Proceedings of the Intl. J. Supercomputer Applications*, 2001.

[13] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari, "Certificate-based access control for widely distributed resources," in *Proceedings of the 8th USENIX Security Symposium*, Washington, USA., August 1999.

[14] M. Lorch and D. Kafura, "Supporting secure ad-hoc user collaboration in grid environments," in *Proceedings of The 3rd Int. Workshop on Grid Computing*, 2002.

[15] R. Lepro, "Cardea: Dynamic access control in distributed systems," NASA Ames Research Center, Tech. Rep., November 2003.

[16] M. Lorch, D. B. Adams, D. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah, "The prima system for privilege management, authorization and enforcement in grid environments," in *Proceedings of the Fourth International Workshop on Grid Computing (GRID03)*, Phoenix, USA, November 2003.

[17] D. W. Chadwick and A. Otenko, "The permis x.509 role based privilege management infrastructure," in *Proceedings of the SACMAT02*, Monterey, California, USA., June 2002.

[18] B. Sundaram and B. Chapman, "Policy engine: A framework for authorization, accounting policy specification and evaluation in grids," in *Proceedings of the 2nd International Conference on Grid Computing*, Denver, Colorado, USA, Nov 2001.

[19] B. Sundaram and B. M. Chapman, "Xml-based policy engine framework for usage policy management in grids," in *Proceedings of the Third International Workshop on Grid Computing (GRID'02)*, Baltimore, USA, November 2002.

[20] K. Keahey and V. Welch, "Fine-grain authorization for resource management in the grid environment," in *Proceedings of the Grid 2002 Workshop*, Ft. Lauderdale, USA, 2002.

[21] K. Keahey, V. Welch, S. Lang, B. Liu, and S. Meder, "Fine-grain authorization policies in the grid: Design and implementation," in *Proceedings of 1st International Workshop on Middleware for Grid Computing*, 2003.

[22] D. C. Verma, S. Sahu, S. B. Calo, M. Beigi, and I. Chang, "A policy service for grid computing," in *Proceedings of the Third International Workshop on Grid Computing*, Baltimore, MD, USA, November 2002.

[23] G. Wasson and M. Humphrey, "Toward explicit policy management for virtual organizations," in *Proceedings of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, 2003.