Federated Access Control and Intrusion Detection for Grid Computing

Li Zhou Clifford Neuman
Information Science Institue
University of Southern California
Marina del Ray, CA 90292
{zhou,bcn}@isi.edu

Abstract

Grid computing enables the sharing of heterogeneous resources within "virtual organizations". Since entities in virtual organizations are independently administrated without global trust, the management of access control and provision of intrusion detection is critical for grid services. Security decisions in grid environments are drawn from multiple domains. Unfortunately the heterogeneity of underlying platforms and the diversity of security mechanisms can make the coordination of such decisions difficult. In this paper, we introduce the Generic Authentication & Access Control API (GAA-API) framework that facilitates interoperability and coordination of policy management for grid services. In particular, we describe the Global Security Token Pool (GSTP) architecture that helps the GAA-API distributed information needed to enforce policy decisions in a distributed environment.

1. Introduction

The grid is a large-scale distributed computing environment that enables the sharing and coordinated use of diverse resources, including computational resources, storage, information and other devices [3]. The grid supports the simultaneous use of a large number of resources, providing dynamic quality of service enforcement, co-allocation, resource discovery and other services for resource management [4][6].

The Grid is built from the resources of a dynamic collection of individuals, institutions and enables the creation of "virtual organizations" [3]. Since these virtual organizations extend across multi-institutional domains, it is usually constructed without global trust. Therefore security issues of authentication, authorization, delegation, certificate management and policy management must be carefully considered to protect the shared resources from all possible security hazards including unauthorized use of resources, misuse of resources, impersonation, etc. Moreover, to cope with platform heterogeneity and administration independency in the Grid environment [5], heterogeneous local security mechanisms (Kerberos, etc.) and diverse forms of local security policies (file system access matrix, configuration file, etc.) should be incorporated.

The Generic Authentication & Access control Application Programming Interface (GAA-API) provides dynamic and fine-grained access control and application-level intrusion detection via simple integration with grid services. The Global Security Token Pool (GSTP) provides the distribution of security policies and the maintenance of security variables. It could help the GAA-API make interoperable decisions over the distributed environment within a virtual

organization. Version 3 of Globus Toolkit (GT3) [8] and its accompanying Grid Security Infrastructure (GSI3) [5] is the first implementation that conforms to the Open Grid Service Architecture (OGSA) [6][7]. However, the current design of GT3 offers little support for access control. Introducing GAA-API and GSTP into the GT3/OGSA framework, we can see the following benefits:

A) The GAA-API provides a straightforward and uniform interface for access control. By integrating the GAA-API into grid services, instead of rewriting and recompiling the source code for access control, administrators can easily customize their specific security requirements by simply writing Extended Access Control List (EACL) policies in plaintext.

B) By implementing each mechanism as a GAA-condition, the GAA-API can easily and seamlessly incorporate multiple grid security mechanisms and local security mechanisms together. The interrelation (AND, OR, etc.) among these mechanisms can be freely regulated by EACL. Moreover, to introduce a new security mechanism into the Gird system, we only need to implement it as a new GAA-condition. This mechanism could dynamically take effect as soon as we add the corresponding condition into the EACL policy.

C) Aided by GSTP, GAA-API can make interoperable access control decisions on the basis of real-time information from other grid services that are located remotely. Thus, advanced access control in a distributed manner such as global quota, dynamic lockdown, etc. can be fulfilled straightforwardly.

D) GAA-API and GSTP support a hierarchy of security policies. Besides the local policy defined for each grid service, host administrators can impose their host-wide policies, and virtual organization administrators can impose its global policy on all grid services within their corresponding domains. Since it's difficult for every individual grid service to define its local policy elaborately enough to protect itself from all potential security threats dynamically, conforming to host-wide policies and global policies could give the grid service a more robust level of security.

E) Besides making access control decisions, GAA-API also support functions such as notification, logging, dynamic intrusion detection & response that are correlated to the authorization phase.

In section 2, we will briefly depict the GAA-API/GSTP architecture for the interoperable access control. In section 3, issues of GAA-API and EACL will be specified. In section 4, topics about GSTP including variable scopes, access restrictions and communication models will be discussed. In section 5, two examples of interoperable access control will be presented to show the strength of GAA-API/GSTP. In section 6, the implementation status and future works will be introduced.

2. Architecture Overview

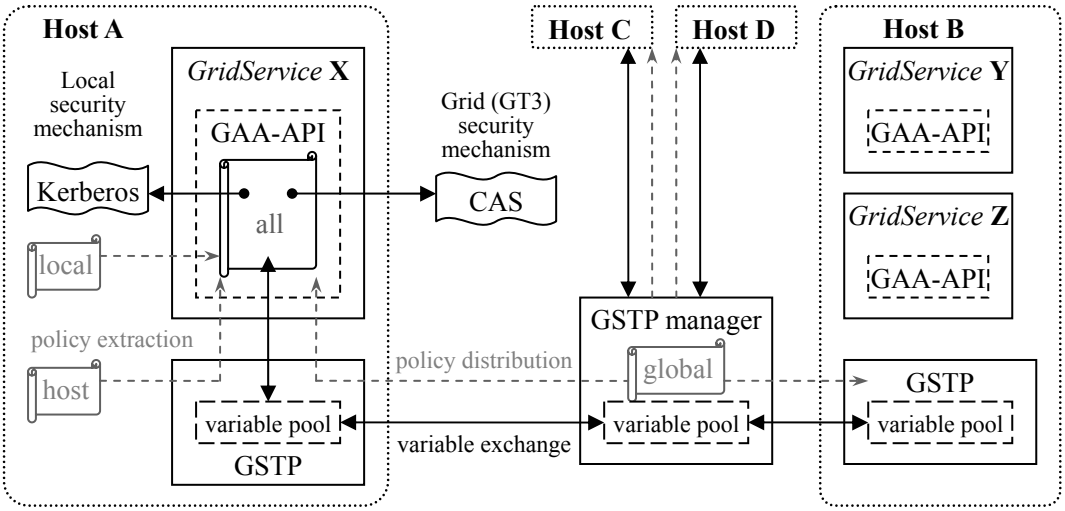

Figure 1: GAA-API/GSTP Architecture

In the Open Grid Service Architecture, a uniform semantic called "Grid Service" is defined. All shared resources and auxiliary agents such as resource management [9], co-allocation [10] and replica location management [11] are virtualized as Grid services [6]. In GAA-API/GSTP architecture, GSTP is also implemented as Grid service. It runs as a daemon process after the system boot-up. If GSTP uses centralized mode instead of peer-to-peer mode for maintenance, one or several GSTP-manager must also start to distribute security policies and maintain security variables. In contrast, the GAA-API acts as an interface library for the Grid services. Only when an instance of Grid service receives a user request and demands an access control decision, is the GAA-API invoked transiently to evaluate corresponding security policies.

As the example in Figure 1 shows, we have multiple hosts administrated independently within a virtual organization. On each host, one instance of GSTP service together with several instances of other grid services are running. At the system boot-up, the GSTP asks its GSTP-manager for the global security policy, and start keeping the consistency of global security variables among all GSTPs and the GSTP-manager. As a grid service instance *X* receives a user request, it invokes the GAA-API with all information that may be useful for the access control decision (such as user IP address, user identity, certificate, etc.). After extracting the global security policy and host security policy from its local GSTP, the GAA-API will combine them with the grid service's local security policy and start evaluating the integrated policy.

During GAA-API's policy evaluation phase, according to the respective demands of security policies, GAA-API may collaborate with other components in the following ways. (1) Down-calls to local security mechanisms for sub-decisions. For instance, ask the local Kerberos mechanisms whether an identity is authenticated). (2) Up-calls to grid security mechanisms for sub-decisions. For instance, ask the Grid Certificate Authority Service (CAS) whether a certain certificate is verified. (3) Asks GSTP for the values of grid security variables and derives sub-decisions from the variables. (4) Updates the values of grid security variables through the local GSTP. (Via reading and writing operations on global security variables, interoperable access control decisions could be drawn over the distributed environment.)

Eventually, GAA-API will integrate all sub-decisions and return its final answer to *X*. If the answer is *YES*, the request to *X* is granted. Otherwise, if the answer is *NO*, the request is rejected or *X* should ask the end user to provide further proof (such as username/password, certificate, etc.) for authorization.

3. Policy Based Access Control

3.1 GAA-API

The Generic Authentication & Access Control Application Programming Interface (GAA-API) provides us with a fine-grained interface of policy-based access control. This framework is based on evaluating the "***policy set***" consecutively, and trying to find a policy that the current request can satisfy. Also, each policy comprises a list of conditions, which is the basic unit of EACL policy. Here, we have two types of policy: ***positive policy*** *GRANTS* a request if all its corresponding conditions are met. In contrast, ***negative policy*** *REJECTS* a request if all its corresponding conditions are met.

"***Conditions***" check access control constraints for time, user identity, user location, etc. imposed by local security mechanisms, grid security mechanism and security variables. Besides passively granting or denying an incoming request, the GAA-API framework also supports real-time inspection of suspicious activity and dynamic reconfiguration of system protection as responses [1]. For this purpose, we introduce a special condition called "action". Instead of reporting *MET* or *UNMET* to GAA-API, "***actions***" perform logging, notification, modifying security variables or other operations that apply to local platforms or the Grid environment.

Most access control systems today are based on the premise that once a request is authorized for an operation, the access will be granted unconditionally for all following operations. This practice can not protect the system from abuses of user privileges during execution [1]. In contrast, the GAA-API framework supports real-time monitoring of the execution process and allows dynamic revocation of granted access rights. This approach introduces a three-phase policy enforcement scheme:

- **Pre-execution phase**: decides whether to grant or deny a request before its execution.
- **Mid-execution phase**: checks whether there are abuses of user privileges or resource consumption exceeding quotas during the execution. If yes, GAA-API will notify the grid service to abort the current job. These checks can be performed either on certain checkpoints or in a periodic manner.
- **Post-execution phase**: decides whether the job is regarded as "successful" or "failed" after its completion.

3.2 EACL

```
#1  <PolicySet xmlns="urn:gaaapi:1.0:eacl:policyset" scope="host">
#2  <Policy type="negative">
#3      <Condition name="check_ip" value="$G{badguy_list}" />
#4  </Policy>
#5  <Policy type="positive" auth="ServiceA" right="read, write, execute">
#6      <Pre><Condition type="check_time" auth="local" value="Mon-Fri,8am-6pm" />
#7          <Action when="both" type="add_to_variable" value="$G{active_ids}/$I{userid}" />
#8          <Condition type="check_variable"> $S{count}<10 </Condition>
#9          <Action when="fail" type="email_notify" link="/etc/email_01 $I{userid}" />
#10         <Action when="succ" type="inc_variable" value="$S{count}" /></Pre>
#11     <Mid><Condition type="check_cpu_usage" value="<90%" /></Mid>
#12     <Post><Action type="append_log" value="$I{userid}'s $I{jobid} finish" /></Post>
#13 </Policy>
#14 </PolicySet>
```

Figure 2: An Example of EACL Policy

An Extended Access Control List (EACL) regulates the security policies for GAA-API. Since the Open Grid Service Infrastructure (OGSI) and Globus Toolkit 3.0 (GT3) use Extensible Access Control Make Language (XACML) as their default format of access control policy [4], here we provide an XML version of EACL that conforms to the schema of XACML [12].

Here's a brief description of EACL schema and semantics:

- **<PolicySet>**: The root tag of EACL policy. Attribute "domain" (optional) regulates whether it is a *global*, *host* or *local* security policy.
- **<Policy>**: Each <PolicySet> comprises an ordered list of <Policy>. Attribute "type" regulates whether it is a *positive* policy (default) or a *negative* policy. Attribute "auth" (optional) and "right" (optional) specify the grid services and requested access rights to which the policy applies.
- **<Pre> <Mid> <Post>**: These tags separate the three phases of policy enforcement. They correspond to pre-execution phase, mid-execution phase and post-execution phase respectively.
- **<Condition>**: Defines a *condition* that applies to this policy. Attribute "type" and "auth" (optional) regulate its corresponding security mechanism. Also, we have three ways to define the constraint of a condition.
  - Defined by the attribute "value", which provides the constraint directly. *(#7, Figure 2)*
  - Defined by the attribute "link", which extracts the constraint from the external file referred to by a pathname or URL. *(#8, Figure 2)*
  - Defined in the domain of <Condition> tag. The constraint here could be specified in XML format. *(#9, Figure 2)*
- **<Action>**: Defines an *action* that may be performed by this policy. Attributes "type", "auth", "value" and "link" are identical to those of <Condition>. Moreover, we have an additional attribute "when" (optional), which has three possible values:
  - **both**: (default value) action is performed no matter the previous condition is MET or

UNMET. *(#7)*

- **succ**: action is performed only when the previous condition is *MET*. *(#9)*
- **fail**: action is performed only when the previous condition is *UNMET*. *(#10)*

3.3 Policy Evaluation

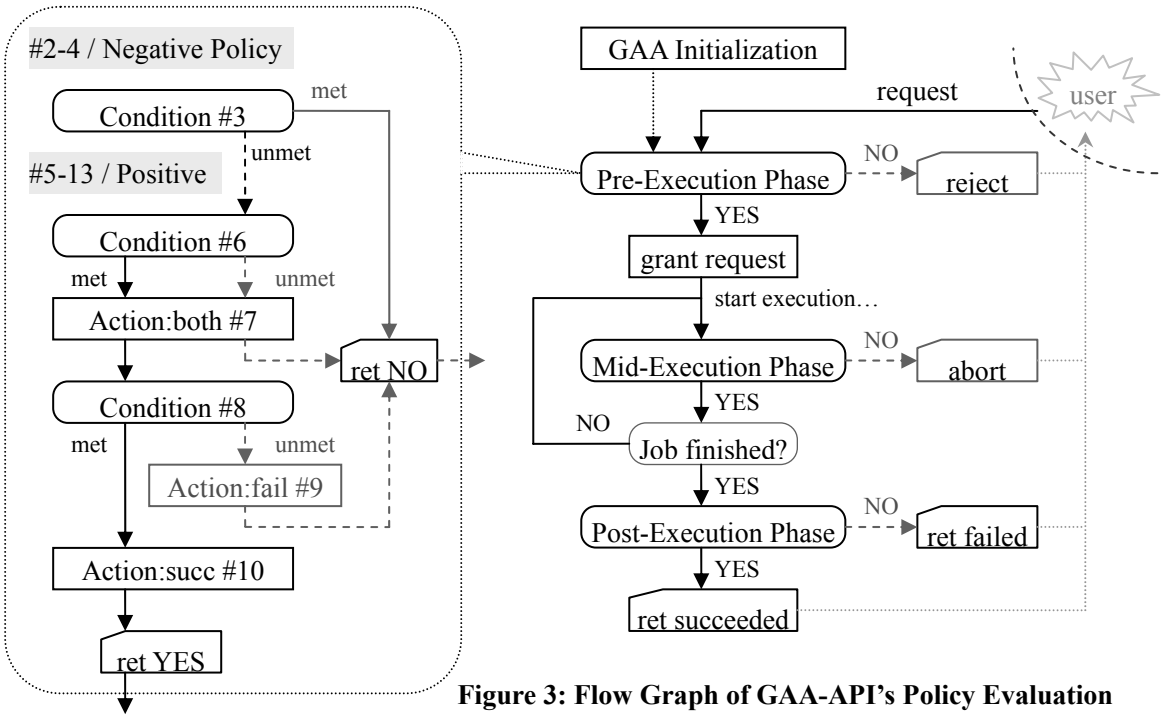

Figure 3: Flow Graph of GAA-API's Policy Evaluation

Figure 3 represents the GAA-API's policy evaluation process, which is based on the EACL policy given in Figure 2. As a GAA-integrated service starts, function *gaa_initialize()* should be called to execute the initialization phase of GAA-API, which includes allocating basic data structures, requesting and caching relevant policy files from the local GSTP daemon.

As the service receives a user request, we will first check whether the cashed policies are current. If not, it is replaced with the latest policies. Next, a GAA-request data structure is constructed, which contains the requested resources and access rights and a list of *Incoming Variables* containing all the user-specific and request-specific information that may be useful to access control decisions. Then, function *gaa_check_authorization()* will be called to start the evaluation of the pre-execution phase (phase-level evaluation).

Within the function *gaa_check_authorization()*, GAA-API will start from the first policy (policy-level evaluation). Within each policy, it also starts from the first condition (condition-level evaluation). For each condition, GAA-API looks up the <Registry> fields defined in global, host or local policy files (Figure 4), finds the corresponding library file and callback-function name, and makes a dynamic call to this referred callback-function. After negotiating with a local or grid security mechanism, the callback-function returns its sub-decision: *MET* or *UNMET* to GAA-API.

```
#1  <Registry xmlns="urn:gaaapi:1.0:registry" scope="local">
#2  <Condition name="check_ip, check_host">
#3      <Entry platform="unix" library="/usr/lib/gaasimple.so" function="gaa_check_ip" />
#4      <Entry platform="win" library="$SYSTEM\gaaip.dll" function="my_check_ip" />
#5  </Condition>
#6  <Condition name="check_time" library="/usr/lib/gaatime.class" function="check_time" />
#7  ……//define the registries of other conditions and actions…
#8  </Registry>
```

Figure 4: An Example of GAA-API Registry

If a condition is returned *UNMET*, the GAA-API evaluation will stop at the current condition and the policy is set as *UNSATISFIED*. Otherwise, if the condition returns as *MET*, GAA-API will go on evaluating the next condition. Only when all conditions within a phase (*Pre-Execution Phase*) are *MET*, is the policy set as *SATISFIED*.

Furthermore, after evaluating each condition, the GAA-API will also evaluate all its corresponding actions (which are defined between the entries of current condition and next condition). However, whether an action is executed or not still depends on its field "when". As "when=fail", the action performs if and only if the current condition is *UNMET*. As "when=succ", the action performs if and only if current condition is *MET*. As "when=both" (default value), the action performs no matter what is the answer from the current condition.

Now we return to the policy-level evaluation. After a policy has all its conditions *MET*, it is therefore set as *SATISFIED*. If the policy is a positive one, the GAA-API will directly answers *YES* to its caller. If the policy is negative, the GAA-API will directly answers *NO* to its caller instead. Otherwise, if the current policy is set as *UNSATISFIED*, the GAA-API will go on evaluating the next policy. However, if all policies are evaluated and set as *UNSATISFIED*, GAA-API will also answer *NO* to the caller.

If the function *gaa_check_authorization()* receives *NO* from the GAA-API, the current request should be rejected. If it receives *YES*, the request should be granted and the Grid Service should start its corresponding job. According to the specific requirements, a Grid Service could invoke function *gaa_exection_control()* to perform mid-execution checks either at certain checkpoints or periodically. If it receives *NO* from the GAA-API, the Grid Service should abort this job as soon as possible. Otherwise the job should go on running. Similarly, after finishing the current job, the Grid Service could invoke *gaa_post_execution_actions()* to finalize access control related operations and decide whether the job has failed (when the GAA-API returns *NO*) or succeeded (when then GAA-API returns *YES*).

4. Global Security Token Pool (GSTP)

The Global Security Token Pool (GSTP) is a Grid service that helps the GAA-API make

interoperable access control decisions. Two major types of tokens are managed by GSTP: security policies and security variables. GSTP maintains these tokens on each host and exchanges them among the distributed hosts within the virtual organization.

4.1 Security Variable: Scope

Security variables are used to share security-related information among GAA-integrated grid services. Similar to the variable in programming languages, a security variable could be an integer number (number of allocated resource), a float number (job execution time), an enumeration (security level: {normal, alert, dangerous}), a list (list of banned IP addresses) or other forms. However, the security variable has its own characteristics. The first factor is: security variable has its own types of scope which applies to the Grid architecture (instead of the structured or object-oriented program). The second factor is: capability of accessing and updating the security variable is restricted by the corresponding security policy.

According to the architecture of the GAA-API and the Open Grid Services Architecture (OGSA) [6], we have the following types of scope available for a security variable.

- Global Security Variable: This type of variable applies to the entire virtual organization. Every GSTP should exchange its local updates with all other GSTP peers, and thus keep the integrity of the global security variable among all GSTPs.
- Host Security Variable: Applies to a specific host. It is uniform to all local grid services, but varies from host to host. Therefore the GSTP only needs to maintain this variable locally. Exchange among GSTP is not necessary.
- Local Security Variable: Applies to a certain Grid service instance (or in other words, an instance of application) only.
- User-Specific Security Variable: Applies to a certain user only.
- Task-Specific Security Variable: Applies to a certain task only. It is especially useful for a task with multiple hosts and multiple services involved.
- Service-Specific Security Variable: Applies to a certain type of service only.

Figure 5 shows all possible combinations of security variable types and their corresponding expression in EACL policy.

Security Variable	Global	Host	Local
Generic	$G{*varname*)	$H{*varname*)	$L{*varname*)
User-Specific	$U{*varname*)	$HU{*varname*)	$LU{*varname*)
Task-Specific	$T{*varname*)	$HT{*varname*)	$LT{*varname*)
Service-Specific	$S{*varname*)	$HS{*varname*)	*not valid*

Figure 5: Types of Security Variable

GSTP can distinguish local, user, task or service security variables by their corresponding service instance ID (Grid Service Reference [6]), user ID, task ID and service URN [6] respectively. Therefore variables that belong to different domains do not interfere with each other.

4.2 Security Variable: Restricted Access

One of the major challenges for Grid is the fact that no global trust exists within the system. This means that although we need to share global security variables among GSTP peers, it may be too insecure to allow all nodes to access and update their values arbitrarily.

Here are two examples of the possible attacks that could be launched against the global security variable. [Example 1]: A malicious host could easily delete itself from the *"banned IP address list"* although others have detected that it is launching dangerous attacks. [Example 2]: If the global quota of resource consumption is imposed on each user, a malicious host could fraudulently report an extra-large number of resource allocations by a certain user and therefore prevent this user from allocating any resource from all other hosts.

Therefore, additional security policies should be introduced to impose restrictions on accessing and updating security variables. If the GSTP receives remote updates that go beyond its restriction, it should ignore this update or confine the change to a permissible scope. Here we represent several types of restriction that may be most commonly used.

- Inaccessible: the GSTP can neither read nor write this variable
- Read-Only: can only read the variable, writing operations are not allowed.
- Condition-Only Read: can not read the actual value of the variable, instead, it can only query the other GSTP holding this value with pre-defined conditions and get *YES* or *NO* as the answer. For example, one could query whether the value of a counter is below 50, or whether an IP address is within the banned-list without knowing the actual content of this counter or banned-list. In addition, the holder can also put constraint on the frequency of queries against the guess of variable content by enumerating all possible values.
- Sum/Average/Min/Max for a Numeric Variable: instead of writing on the same numeric variable, local shares contributed by every GSTP are still maintained. However, when we read this variable, the GSTP will provide us with the value integrated by function sum(), average(), min() or max().
- Per-Host Range for Numeric Variable: besides keeping local shares as we write, we also impose a range "[*lower*, *upper*]" that each share can not exceed. This restriction is especially useful for global quota. For example, if our total quota is 15, and we mandate that the shares provided by host A/B should be within [0,5] and the shares provided by other hosts should be within [0,3], then the attack depicted in Example 2 could be effectively confined.
- Append-Only (or Remove-Only) for List Variable: append-only means that the GSTP can only add new elements into a list, but is unable to update or remove elements from it. Remove-only is just the reverse. These two restrictions are especially useful to protect the attack depicted in Example 1.

4.3 Design Issues for GSTP

The first critical design issue is the communication model used by the GSTP. Three major options are available here: a centralized model, a distributed model and a peer-to-peer model.

In the centralized model (Figure 1), global security policies are handed out by a GSTP-manager. All updates on global security variable are at first submitted to the GSTP-manager and then forwarded to other GSTP peers via this manager. Moreover, replication could also be introduced into the centralized model to improve availability.

The centralized model is straightforward and easily administrated. It fits well with the distribution of global security policies because they are not expected to change very frequently. However, for the exchange of global security variables, if there are a great number of GSTP nodes involved and the frequency of variable updates are expected to be high, a centralized GSTP-manager will become a bottleneck.

In this case, the distributed model or the peer-to-peer model should be considered instead. In the distributed model, each GSTP is responsible for managing and forwarding a small portion of variables only. In the peer-to-peer model, GSTP will send local updates to all other GSTP peers directly. Since in a Grid system, no GSTP node is guaranteed to be online, availability and integrity should be carefully considered when implementing the distributed or the peer-to-peer model.

The second critical design issue is how to keep the integrity of every global security variable. In the security policy file, GSTP should allow policy-makers to choose which mode is more appropriate for a certain variable. Principally, we have the following integrity modes available:

- Push Mode (default): when an update for a global security variable occurs, GSTP will exchange the new values with other GSTP peers. This mode is appropriate when reading operations occur more frequently than writing operations.
- Poll Mode (default): the GSTP extracts the current value from the variable-holder as long as each reading operation occurs. Then for each writing operation, the GSTP only needs to submit the updated value to the variable-holder. This mode is appropriate when the variable is volatile, but reading operations occur infrequently.
- Periodic Poll Mode (default): In contrast to the "*Poll Mode*", the GSTP only extracts updated values periodically. This mode is appropriate only when the variable is both volatile and frequently read. But we don't require its exact current value to be evaluated all the time, e.g., variables for the average usage of storages and CPUs.

5. Examples

As we discussed in the previous sections, the GAA-API and GSTP can cooperate in providing interoperable access control for the Grid system, by flexibly and seamlessly integrating local and grid security mechanisms altogether. To show the strength of GAA-API/GSTP, here we provide two demo scenarios. The first example is global quotas for resource allocation. The second example is distributed denial of service (DDoS) detection with dynamic lockdown.

5.1 Scenario One: Global Quota

Assume that we have 26 hosts: "A"-"Z" within the virtual organization, we try to impose a global quota on computational resources throughout the virtual organization. Here's the security policy for this scenario. #1-9 defines the policy which is evaluated by GAA-API. #10-16 regulates the variable restrictions which are inspected by GSTP.

```
#1  <PolicySet xmlns="urn:gaaapi:1.0:eacl:policyset" scope="global">
#2  <Policy type="positive" auth="CPU" right="execute">
#3  <Pre>   <Condition type="check_grid_id" auth="X.509" value="IN $G{cpu_userlist}" />
#4          <Condition type="check_variable" value="$U{proc_count}<20" />
#5          <Action when="fail" type="append_log" link="$I{user} denied on $I{now}" />
#6          <Action when="succ" type="inc_variable" value="$U{proc_count}" />   </Pre>
#7  <Post>  <Action type="inc_variable" value="$U{proc_count}" />  </Post>
#8  </Policy>
#9  </PolicySet>
#10 <VariablePool xmlns="urn:gaaapi:1.0:gstp:variablepool" scope="global">
#11     <Variable name="cpu_userlist" type="list" restriction="condition-only-read"
#12         expression="IN $" mode="poll" />
#13     <Variable name="proc_count" type="integer/sum" restriction="per-host-range">
#14         <Host name="A,B,C" range="[0,5]" /><Host pattern="[D-Z]" range="[0,3]" />
#15     </Variable>
#16 </VariablePool>
```

Figure 6: Security Policy of Global Quota Scenario

To grant the request for a computational resource (Figure 6: #2), firstly the GAA-API should ensure that the user sending this request is within the global variable "cpu_userlist" (#3) regulated by the administrator. Then, the GAA-API should check that no more than 20 processes are allocated simultaneously for each user throughout the virtual organization (#4). If the request is granted, the GAA-API will increase the per-user counter "proc_count" by one (#6). And, after the job is complete, GAA-API will decrease the counter "proc_count" by one (#7). These two actions ensure that the variable "proc_count" keeps the number of processes allocated by each user throughout the virtual organization.

Besides the policy evaluated by GAA-API, we also have the following variable restrictions imposed by GSTP. First, only the GSTP-manager knows the content of "cpu_userlist". Other GSTPs can only inquire whether a certain user is on this list or not (#11-12). Moreover, to protect the system from the "fraudulent report attack", host A, B & C can report no more than 5 allocated processes to "proc_count", other hosts are allowed to report no more than 3 (#13-15).

5.2 Scenario Two: DDoS Detection & Dynamic Lockdown

Assume that we have an arbitrary number of hosts providing HTTP web-server services within a virtual organization. Each host checks its per-user request frequencies locally (statistics on global request frequencies are impractical because of the high cost). And local DDoS suspicions are dynamically synthesized to the global decision of DDoS detection and response.

```
#1  <PolicySet xmlns="urn:gaaapi:1.0:eacl:policyset" scope="global">
#2  <Policy type="positive" auth="WebServer" right="head,post">
#3  <Pre>    <Condition type="check_ip" value="128.9.0.0/16 AND NOT $G{banned_ip}" />
#4           <Condition type="check_req_freq" value="$HU{req_freq}<=20/min" />
#5           <Action when="fail" type="inc_variable" value="$U{suspicion}" />
#6           <Condition type="check_variable" value="$U{suspicion}<3" />
#7           <Action when="fail" type="add_to_variable" value="$I(ip)->$U(banned_ip)" />
#8           <Action when="succ" type="calc_req_freq" value="$HU{req_freq}" />   </Pre>
#9  </Policy>
#10 </PolicySet>
#11 <VariablePool xmlns="urn:gaaapi:1.0:gstp:variablepool" scope="global">
#12     <Variable name="suspicion" restriction="per-host-range, inc-only, valid-period"
#13         scope="user" type="integer/sum/ts" range="[0,1]" invalid_after="30min" />
#14     <Variable name="banned_ip" type="list" restriction="add-only" />
#15 </VariablePool>
```

Figure 7: Security Policy for DDoS Detection & Dynamic Lockdown

To grant a request on the web server (Figure 7: #2), the GAA-API will first check whether the IP address of the requested user is within 128.9.0.0/16 and doesn't belong to the global security variable "banned_ip" (#3). Then, the GAA-API checks if the per-user request frequency is less than 20 times within the past minute (#4) and has the frequency recalculated whenever a request is granted (#8). If such local frequency is beyond the quota, the GAA-API will increase the value of the global per-user variable "suspicion" (#5). The next condition checks whether we have less than 3 individual hosts reporting DDoS suspicions within the past 30 minutes (#6). If such condition is not approved, the GAA-API will add the IP address launching the DDoS attack into "banned_ip" (#7) and therefore perform a dynamic lockdown which prohibits this IP address from accessing any web server (#3) within the entire system.

Furthermore, GSTP imposes that that each host can only contribute a share of either 0 or 1 to the variable "suspicions", and every updating operation is valid for 30 minutes only (#12-13). Also, we have the regulation that a GSTP can only add elements to the variable "banned_ip", but is unable to remove any element from it. (#14)

6. Future Work

At the current stage, we already have the GAA-API available, which supports dynamic condition registration, three-phase policy enforcement, etc. It has already been successfully integrated into the Apache Web Server [2][15], Secure Shell [14][16] and IPSecurity [17]. But we still need the following improvements to make it compliant with the Grid environment: Firstly, the current version of GAA-API only supports EACL policy in plaintext format. We should further migrate to the XACML-compatible format of the EACL policy to conform to the framework of OGSA [6][7] / Globus Toolkit 3 [8]. Secondly, we should provide new GAA-API libraries that are used

for communicating with the local GSTP service. Thirdly, we only have a C version of GAA-API which can run on Linux and Sun Solaris at present. To satisfy higher platform heterogeneity and better compatibility with Globus Toolkit 3, a Java version of GAA-API may be necessary.

Moreover, the Global Security Token Pool (GSTP) is still not available at present. We need to implement GSTP service, which contains various applicable scopes, access restrictions, 3 basic integrity modes, and 3 basic communication models for security variables together with the distribution mechanisms for security policy in our future work. Moreover, we should provide programming interfaces that allow users to customize their own variable restriction, integrity mode, or communication model easily.

References

[1] Tatyana Ryutov, Clifford Neuman, The Specification and Enforcement of Advanced Security Policies, *In Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002), June 5-7, 2002, in Monterey, California.*

[2] Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou, Integrated access control/intrusion detection for securing web server, *IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 9, September 2003.*

[3] I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations**.** *International J. Supercomputer Applications*, 15(3), 2001.

[4] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A Security Architecture for Computational Grids. *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pp. 83-92, 1998.

[5] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Cajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, S. Tuecke, Security for Grid Services. *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, IEEE Press, June 2003.

[6] I. Foster, C. Kesselman, J. Nick, S. Tuecke, The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002

[7] S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling, Open Grid Services Infrastructure (OGSI) Version 1.0. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, Global Grid Forum Draft Recommendation, 6/27/2003.

[8] Globus Toolkit 3, http://www.globus.org

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

[10] K. Czajkowski, I. Foster, and C. Kesselman, Resource Co-Allocation in Computational Grids. *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pp. 219-228, 1999.

[11] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B, Schwartzkopf, H, Stockinger, K. Stockinger, B. Tierney, Giggle: A Framework for Constructing Sclable Replica Location Services. *Proceedings of Supercomputing*

2002 (SC2002), November 2002.
[12] Simon Godik, Tim Moses, Extensible Access Control Markup Language (XACML) Version 1.0
[13] Don Box, Francisco Curbera, etc., Web Services Policy Framework (WS-Policy) Version 1.1
[14] Tatyana Ryutov, Clifford Neuman, and Dongho Kim, Dynamic Authorization and Intrusion Response in Distributed Systems, *In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III), Washington, D.C. April 22-24, 2003.*
[15] Apache Web Server, http://www.apache.org
[16] Open SSH, http://www.openssh.com
[17] IP Security, http://www.freeswan.org