Entity Management and Security in P2P Grid Framework

T. N. Ellahi, B. Hudzia, L. McDermott, T. Kechadi, A. Ottewill

Parallel Computational Research Group,
School of Computer Science and Informatics,
University College Dublin, Belfield, Dublin 4, Ireland
`tariq.ellahi@ucd.ie, benoit.hudzia@ucd.ie`
`liam.mcdermott@ucd.ie, tahar.kechadi@ucd.ie, Adrian.Ottewill@ucd.ie`

Abstract. During the last decade there has been a huge interest in Grid technologies, and numerous Grid projects have been initiated with various visions of the Grid. While all these visions have the same goal of resource sharing, they differ in the functionality that a Grid supports, characterization, programming environments, etc. In this paper we present a new Grid system dedicated to deal with data issues, called DGET (Data Grid Environment and Tools). DGET is characterized by its peer-to-peer communication system and entity-based architecture, therefore, taking advantage of the main functionality of both systems; P2P and Grid. DGET is currently under development and a prototype implementing the main components is in its first phase of testing. In this paper we gives description of two main components of DGET: Entity Management and Security subsystem.

1 Introduction

In recent years, Internet-scale systems have been developed and deployed to share resources at a very large scale across the traditional organisational boundaries. The need for constructing such systems was motivated by the increasingly complex requirements of modern applications from diverse disciplines. Such global scale systems provide opportunities to harness idle resources which are distributed and heterogeneous. Another benefit offered by such systems is that they allow coordinated use of resources from multiple organisations. Thus, these wide-area systems may span multiple organisations and form virtual organisations on top of the existing organisational hierarchies. Two such systems exploiting these views include Grid and Peer-to-Peer (P2P) systems. Grid and P2P have seen a rapid evolution and widespread deployment. The two technologies appear to have the same final objective, pooling and coordinating large sets of distributed resources[1]. During the last few years various projects have been undertaken to try to merge these two complementary approaches of these technologies, such as OurGrid[2]. Also various modifications to the Globus toolkit[3] have been proposed to include P2P technology and thus improving the discovery system[4].

Typically, Grid systems are designed to run applications with intensive computing and storage needs across the traditional organisational boundaries[5–7]. They are characterised by their sophisticated resource management and data transfer components. P2P systems on the other hand were mainly designed for resource sharing, mostly files. Therefore, the focus of P2P systems is on providing sophisticated resource discovery capabilities. Both approaches have their own advantages and disadvantages.

In this paper we describe DGET (Data Grid Environment & Tools). DGET is a P2P based grid middleware. This paper explains the functionality of two main components of the middleware: EntityManager and Security. Details of DGET architecture and other components can be found in [8][9][10][11]. The rest of the paper is structured as follows: Related work is described in section 2. Section 3 and 4 explain general overview and an intrduction to DGET architecture respectively. Details about Entity Management are given in section 5 and Security subsystem is explained in section 6. The paper concludes in section 7.

2 Related Work

DGET is P2P Grid middleware and employs techniques from both fields. DGET should be compared to other midlewares adopting the P2P Grid approach. The following paragraphs describe how DGET is distinguished from existing solutions.

DGET and Grid Middleware: A number of grid midleware has been developed and used. A wide range of systems have been developed. Some of these focus on providing the core middleware services while other programming frameworks are built on top of these middleware systems and provide high-level application development functionalities. Globus, Legion and UNICORE are the most notable grid middlewares. The Globus Toolkit is the most widely used middleware. DGET has some distinct characteristics. First, existing grid middlewares adopt a manual and static topology whereas DGET is based on dynamic, self-organizing topology borrowed from the decentralised P2P systems. Other distinguishing DGET features include a decentralized P2P style resource discovery and fine grained access control. Existing grid systems depend on specialized central servers to maintain information about shared resources. DGET, on the other hand adopts the P2P style decentralized resource discovery approach and thus doesn't rely on any specialized servers.

On the security front, Globus possess an extremely powerful security system but it has considerable management overhead. All the users are required to have individual accounts on the machines before they can use the resource. This situation is applicable if there are a limited number of participants. In a situation where a very large number of users are present this technique would become very cumbersome. DGET on the contrary doesn't require users to have individual user accounts on the resources. DGET's security mechanism is based on an extended Java security model. Other aspects where DGET security differs from Globus are the fine-grained access control policies and the resource quota control. DGET uses XACML[12] to define fine-grained access control policies.

DGET and Hybrid Systems: Some system designers have tried integrating both P2P and Grid approaches to come up with a system which enjoys the benefits of both grid and P2P systems[4]. This section compares DGET's approach with such hybrid P2P Grid systems. Our Grid is one such P2P Grid middleware. OurGrid[2] bears many similarities with DGET but has some differences as well. Our Grid lacks sophisticated resource discovery solutions present in DGET. Another difference between DGET and OurGrid is migration support. DGET supports strong transparent migration of applications but OurGrid does not.

3 DGET Overview

As described in the related work section, there are a few systems that have combined the concepts from both Grid and P2P systems. Such hybrid systems are called P2P Grids. DGET adopts the same approach and exploits the advantages of both systems and provides an integrated environment for manipulating and analyzing very large data sets.

3.1 DGET Objectives

We have set the following high-level objectives for DGET middleware.

Uniform Management Interface: Resources in DGET systems are represented through a standard and uniform interface. This approach helps in masking the intra-resource heterogeneity. Users don't have to master the entire heterogeneous interface. New resources can be seamlessly added to the system.

Simplicity & Ease of Use: Grid users are typically non-technical, therefore, it is imperative that grid middleware should be simple and easy to use. DGET should tackle the low-level complexities and make it simple for the grid users to use and manage.

Fault Tolerance: In a large scale grid system, faults are not an exception but a norm. DGET should be able to manage the survive system failures transparently without degrading the application performance.

Scalable Architecture: DGET architecture should be scalable to accommodate thousands or even millions of users, resources and data sets. DGET topology must be decentralized and dynamic as centralized architecture result in poor scalability of the system

3.2 DGET Concepts

Entity: An Entity is a network enabled discrete unit of abstraction that provides some functionality to its users. Entity can take many forms e.g. a remote computation, a remote object, a server that processes user requests etc. The Concept of an entity is akin to a process in the operating systems. An Entity is a mobile element that can move around on different nuclei. An Entity is composed of two parts, a system provided Shell and user provided Ghost. Definitions of these are given below.

Shell The Shell is the system provided control part of the entity. Shell exposes a management interface through which entities can be manipulated. Shell is attached to the programmer provided Ghost when an entity is created.

Ghost The Ghost represents the programmer provided part of an entity. Ghost implements the actual logic of the functionality.

Nucleus The Nucleus is the kernel of the system. It Provides basic services like lifecycle management, communication, security etc. to entities.

Connector Transport protocol agnostic communication medium provided to entities for communicating with each other. Connector is a polymorphic artifact that supports a rich set of interaction models between the entities. Connector is a high level construct which shields programmers from low-level connection setup related operations. Another distinguishing feature of connector is that it is a restorable communication medium which plays a key role in the entity migration process.

4 DGET Architecture

In this section we will give an overview of the architectural components. Detailed description of these components is given in their respective sections. The purpose of this section is to give an overview about how all the components are structured and organised. Figure 1 shows a diagrammatic overview of the system. The DGET system is composed of three logical layers. The Following is a brief description each layer and the components residing in that layer.

Core Layer This layer provides basic services to the entities executing in the nucleus. These basic services include communication facilities, lifecycle management and security.

Facilitation Layer This is the second layer in the system. It facilitates execution of the entities by providing them certain services. The components residing at this layer are also modeled as entities. Entities residing at this layer are called System entities. System entities use the services provided by the core layer. Certain components from the Core layer are modeled as system entities as well. Therefore, in the diagram, Security and EntityManager components span both Core and Facilitation layers. The following entities are located at this layer.

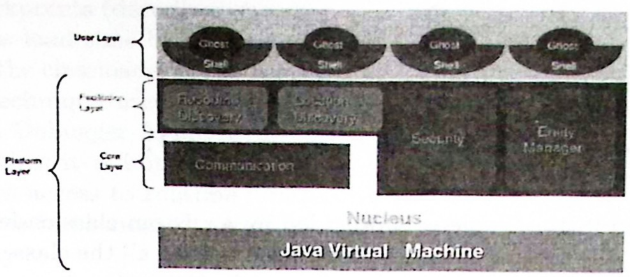

Fig. 1. DGET Architectural Compenents

- **Entity Manager Entity** This entity provides lifecycle management services. This entity to instantiate new entities or manipulate existing entities.
- **Policy Entity** This entity serves as the policy repository of the nucleus. Access control and other management related policies are maintained in the policy entity.
- **Resource Discovery Entity** This entity implements the DGET resource discovery component. Resources and services provides by other entities are discovered through resource discovery entity.

User Layer This is the top most layer in the system. Entities developed by the users and deployed into the system reside at this layer. Entities located at this layer provide user implemented functionality to the users.

5 Entity Management

Entity Creation & Isolation The `EntityManager` entity is responsible for initiating the creation of a user entity in the Nucleus. As described previously, the `EntityManager` functionality is exposed as a System Entity in the Nucleus. The `EntityManager` Entity (EME) publishes its existence along with the characteristics of the host so other entities can locate EMEs according to their requirements. In order to access the local EME running in the same Nucleus, entities can use `EntityContext`. EME creates a shell and passes it the system parameters required to load a ghost. These system parameters include a `GhostLoader` reference, `ThreadGroup` reference and information about the Ghost to be instantiated. Shell uses these parameters and instantiates the Ghost. After successful instantiation of the Ghost, the Shell calls the `setEntityContext()` method on the Ghost class passing in the `EntityContext` object. The Shell also passes an instance of itself to the Ghost. The Ghost can use this instance to invoke lifecycle management operations on itself. The `EntityContext` class is the medium ghosts can use to access system services supported by the Nucleus.

```
public class EntityContext {
  public Connector getEMEntity();
  public Connector getRDEntity(){};
  public Shell getShell();
  public NucleusInfo getNucleusInfo();
  public Resource[] getResourceLimits();
  public Resource[] getResourceConsumption();
}
```

Entity isolation in the Nucleus is provided by a custom classloader called `GhostLoader`. A separate GhostLoader is used to load all the classes belonging to an entity thus providing a separate namespace for the entity classes. GhostLoader associates a security context with the entities classes. This security context is used during its execution to take the access control decisions. Communication between entities is done through connectors. `GhostLoader` has other functions in DGET beside providing separate namespaces for entity classes. These include instrumenting entity bytecode to support soft termination, transparent migration and resource control etc.

Soft Termination Entity termination means killing all the threads an entity might have created during its execution. Sun has declared the thread termination methods as potentially unsafe[13] and deprecated them. Another approach should be adopted to terminate all the threads belonging to an entity. DGET uses the following approach for soft termination of entity threads:
An `Execution` class is introduced. This `Execution` class has a flag indicating the execution state of the entity. During the execution, all entity threads call the `check()` method periodically. If execution flag is RUNNING, `check()` method returns silently but if execution state is TERMINATED, the `check()` method throws
`EntityTerminatedException`. During the loading process, entity classes are instrumented with this execution checks. All the methods are also instrumented with `try/catch` blocks. The `catch` catches the `EntityTerminatedException` exception and re-throws this exception to propagate it down the thread stack. Entity classes are not allowed to catch this exception. During the classloading and instrumentation process, entity class files are scanned to find exception handlers for the `EntityTerminatedException`. This scanning ensures the malicious programmer don't catch this error in order to avoid the entity termination.

5.1 Migration Support

One of the distinguishing features of DGET is strong migration support in a transparent manner. This section describes implementation details of migration support in DGET.

Implementation Techniques The deciding factor in choosing these methodologies were the requirements of portability of the solution, minimal space and time overhead. Code blocks injected into entity classes through bytecode unstrumentation perform different functions like program counter restoration and execution checkpoints (described shortly). The bytecode instrumentation is performed at class load time by a custom classloader. Bytecode instrumentation is performed by the classloader using the Byte Code Enginerring Library(BCEL)[14]. The Second technique used for capturing and restoring execution state is the Java Platform Debugger Architecture (JPDA). JPDA is part of the JVM specification and thus it is implemented by every standard JVM implementation. JPDA provides access to runtime information of the JVM including the thread stacks. JPDA is implemented purely in Java so our migration solution doesn't lose portability

Migration Enabling Features These paragraphs explain the features that enable transparent strong migration in DGET. In order to perform migration at an arbitrary point, values on the operand stack must be saved and restored during the entity restoration process. JPDA doesn't expose any methods to access the values currently present on the operand stack. Initiating migration at such point might result in loss of data from the operand stack. One solution could be to insert checkpoints in the code at locations where execution is not in the middle of a source code level instruction. Migration requests should be delayed till the execution reaches any such checkpoint. Execution checkpoints are inserted as the first instruction in every method and in all the loops in every method.

Another DGET feature to support multi-threaded migration is Mobile Monitors. Java provides multi-threading support in the form of `synchronized` methods and code blocks. A monitor is associated with each java object by JVM and before entering a `synchronized` method or code block, a thread has to acquire the monitor associated with the object. Monitors associated with java objects are maintained and hidden inside the JVM. These monitors are not serilizable and thus are not transported with the serialized objects. Mobile monitors preserve the lock state upon migration. During the class loading process, class files are instrumented to replace Synchronized methods and code blocks. A Mobile monitor is associated with a class that requires synchronized access.

Migration Process

Entity Suspension: The migration process is initiated when the `export()` method is invoked on the Shell. Execution checkpoints discussed in the previous section are used to halt the execution of the entity The `export` method calls the `suspend()` method on the associated `Execution` class. As a result, execution of all the threads is blocked on the next execution checkpoint.

State Capture The `StackFrame` class from JPDA represents a method call on the thread stack. The `StackFrame` class gives access to the values of local variables and the program counter. Calling the `visibleVariable()` method on the `StackFrame` class returns a list of all the variables accessible till the point of execution in the method code. Execution state of all the entity threads along with the mobile monitors and Execution class is saved in a serializable format and transported to the destination for reincarnation of the entity.

State Restoration On the destination nucleus, entity state is restored by calling the `import()` method of the shell. Saved image of entity's execution context is passed as a parameter to the `import()` method.To reestablish the execution state of a thread, its method stack must be rebuilt. To do this, all the methods are called in the order they were on the stack before execution was suspended and migration was initiated. Event handlers can be set that are called when method entry/exit event occurs. When a method entry event occurs, such event handlers restore the values of local variable of the method from the saved execution image. After restoring local variables execution should continue from the code position which is the method invocation for the next method on the stack. Doing so will ensure the instructions already executed are skipped and restoration of the next method on the stack frame begins and proceeds in the same manner. After restoring all the threads to the state they were before the migration was initiated, the `resume()` method on the `Execution` class is called. This method sets the execution status flag to RUNNING and notifies all the threads blocked on this class. Execution proceeds normally afterwards.

As mentioned in the previous paragraph, after restoring local variables, execution should continue from the code position which is the method invocation of the next method on the stack. No mechanism is available in JPDA to set the value of the program counter register to this code position. This problem is solved by maintaining an artificial program counter (APC) which represents an index of method invocations in the method. This APC is incremented after every method invocation instruction. This APC is used in conjunction with a `tableswitch` bytecode instruction which branches the execution according the value of the APC. This `tableswitch` and APC increment instructions are added during the instrumentation procedure. `tableswitch` is added at the beginning of each method and defaults to the original starting code position of the method code.

6 Security

In opposition to grid systems, no centralized servers are present in P2P thus security should not rely on the presence of a centralized server to store and process security related information. All the security related decisions should be made in a decentralized manner making the system scalable. The security model in DGET is designed keeping in mind the P2P system characteristics. The following are the important features of the DGET security model:

- Distributed low-overhead identity based authentication mechanism
- Policy based fine-grained access control
- Distributed security policy management
- Fine grained need-to-act based permission delegation

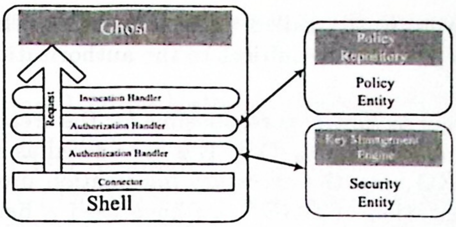

Fig. 2. Security Handlers

6.1 DGET Security Architecture

DGET employs several techniques and components to provide a secure execution environment for entities. This section provides a brief overview of the DGET security architecture. The description of the security system architecture is as follows:

Security Policy-Aware Resource Discovery DGET is equipped with a sophisticated P2P style resource discovery system[8][9]. It is important to enhance resource discovery with security policies so that only those resources are discovered which the user has access to thus increasing system efficiency and productivity. Access control policies are advertised along with the resources so the resource discovery system can analyze this policy during the resource discovery process.

Security Handlers The Shell being the control part of an entity, is the most logical place to perform security related operations. The Shell is equipped with a set of security handlers which process the request to apply. security functionality. These handlers are structured in the form Chain of Responsibility (CoR) design pattern. The following two handlers perform security operations:

Authentication Handler This handler performs the authentication and establishes identity/attributes which can be used during the authorization decision process. Details of the authentication mechanism are described in the following sections

Authorization Handler After the successful completion of the authentication process by the authentication handler, user attributes are extracted from user credentials and the request is passed on to the Authorization Handler (AH). This handler carries out the authorization decision process.

Policy Repository Access control information is specified using policies which are updated dynamically. There are multiple levels of security polices specified by different users according to their roles in the system. Policy information is maintained in a separate Entity called the Policy Entity. The Policy Entity is a System Entity and thus it is also subject to the authorization.

Security Entity Security Entity is responsible for the creation and verification of certificates, keys and signatures. The SE is considered as a Trusted Authority (TA) that is valid PKG with the necessary information proving its validity. In the current implementation of DGET, we have used a hard coded certificate that exists in all nuclei to ensure a high level of security for the system. The SE plays the role of Keystore for all the other Entities within the Nucleus Since this Entity holds the keys and handles signature verification, we implemented a cache system for frequent authentication and signature verification.

6.2 Authentication Mechanisms

We have designed Identity Based Cryptographic (IBC) solution to handle the authentication of DGET. We were inspired by various solutions that appeared lately in[15, 16, ?]. It provides an easy way to manage keys and the benefits from the ID-based approach include:

– Automatic revocation via expiry of time-limited identifiers.
– Reduced round trips if the user can predict delegation requests.
– Reduced bandwidth.
– Similar computational costs.
– Trivial computation of proxy key pairs (RSA key pair generation replaced by elliptic curve multiplication).
– Replication of existing GRID security capabilities.
– Possibility of providing Signencryption scheme.

The IBC system we decided to implement is a variation of the SOK-IBS[ref]

SOK-IBS scheme: This subsection gives formal definitions of presumed hard computational problems on which the SOK-IBS relies.

Bilinear Maps Let G be a cyclic additive group generated by P, whose order is a prime q. Let V be a cyclic multiplicative group of the same order. We use Weil or the Tate pairing ($\hat{e} : G \times G \rightarrow V$) over supersingular elliptic curves or Abelian varieties over finite field because they can provide admissible maps over cyclic groups satisfying those properties[17]:

– Bilinearity:
For any $P, Q, R \in G$, we have $\hat{e}(P + Q,R) = \hat{e}(P ,R)\hat{e}(Q ,R)$ and $\hat{e}(P,Q + R) = \hat{e}(P,Q)\hat{e}(P,R)$.
In particular, for any a, b $\in \mathbf{Z}_q$, $\hat{e}(aP, bP) = \hat{e}(P, P)^{ab} = \hat{e}(P, abP) = \hat{e}(abP, P)$.
– Non-degeneracy:
There exists $P,Q \in G$, such that $\hat{e}(P,Q) \neq 1$.
– Computability: There is an efficient algorithm to compute $\hat{e}(P,Q)$ for all $P,Q \in G$.

Diffie-Hellman problems Consider a cyclic group $\mathbf{G}_1$ of prime order q.

- The **Computational Diffie-Hellman problem** (CDH) in $\mathbf{G}_1$ is, given $\langle hP, aP, bP \rangle$ for un- known $a, b \in \mathbf{Z}_q$, to compute $abP \in \mathbf{G}_1$.
- The **one more CDH problem** (*1m-CDH*) is,
  given $\langle P, aP \rangle \in \mathbf{G}_1$ for an unknown $a \in \mathbf{Z}_q$, and access to a target oracle[18] $T_{\mathbf{G}_1}$ returning randomly chosen elements Y_i *in* $\mathbf{G}_1$ (*for* $i = 1; \dots ; q_t, q_t$) being the exact number of queries to this oracle) as well as a multiplication oracle.
  $H_{\mathbf{G}_{1,a}}(.)$ answering aW *in*$\mathbf{G}_1$ when queried on an input W *in*$\mathbf{G}_1$, to produce a list $((Z_1; j_1), \dots, (Z_{qt}, j_{qt}))$ of q_t pairs such that $Z_i = aY_{ji}$ *in*$\mathbf{G}_1$ for all $i = 1,\dots, q_t$, $1 \leq j_i \leq q_t$ and $q_m < q_t$ where qm denotes the number of queries made to the multiplication oracle.

Scheme The modified SOK-IBS scheme was proven to be as secure as the one-more Diffie-Hellman problem [19]. This scheme is made of four operations:

- Setup:Given a security parameter k, the Private Key Generator (PKG) selects groups G_1 and G_2 of prime order $q > 2^k$, a generator P of G_1, a randomly chosen master key s $\in Z_q$ and the associated public key $P_{pub} = sP$. It also selects cryptographic hash functions of the same domain and range $H_1, H_2 : 0, 1 \rightarrow G_1{}^*$. The public parameters of the system are:
  params $= (G_1, G_2$ ê, P, Ppub,$H_1, H_2)$
- KeyGen:Given the ID of a user, the PKG computes
  $Q_{ID} = H_1(ID) \in G_1$ and the associated private key
  $d_{ID} = sQ_{ID} \in G_1$ that is transmitted to the user.
- Sign:In order to sign a message M,
  - Pick a random integer $r \in Z_q$ and compute
    $U = rP \in G_1$. Then $H = H_2(ID,M,U) \in G_1$.
  - Compute $V = d_{ID} + rH \in G_1$ where + indicates addition operation on the group G_1.

  The signature on M is the pair = (U, V) $\in G_1 \times G_1$.

- Verify:To verify a signature = (U, V) $\in G_1 \times G_1$ on a message M, the verifier first obtains the ID of the signer and computes $Q_{ID} = H_1(ID) \in G_1$. The verifier recalculates $H = H_2(ID,M,U) \in G_1$.
  The signature is accepted if $\hat{e}(P, V) = \hat{e}(P_{pub}, Q_{ID})\hat{e}(U,H)$ and is rejected otherwise.

The multi-authority scalable DGET Authentication systems: As noted in [19] the use of a random seed to handle unlinkability concerns allows the sender and receiver to have different PKG since no pairing is involved with the

receiver's private key (and respectively with the sender's one). This allows us to provide signature and hence authentication capabilities as long as the public keys of the involved PKGs are trusted. Such specific functionality is used to create a hybrid solution between a full identity based solution like a hierarchical ID based one and traditional PKI system. This solution allows more flexibility than traditional PKI or HIBE while reducing the overall network load and computational overhead on the system. In the next section we will describe how we implemented such a system.
Every single Entity, Nucleus and user has its own ID. So before becoming part of DGET every element must register its identity with a TA. Upon a successful registration of ID, the TA will issue the corresponding private key. Every Nucleus possess a Security Entity, so no remote secure communication channel is required to be open and the authentication process is purely local. Since the public key is simply the identity string, this allows a more fine grained control of the key management by adding more information to the identity string such as: a validity period ,delegation characteristic, security domain restriction,etc... The validity period depends on the type of element registered. While a Nucleus might get a long validity period an Entity will get a shorter one corresponding to their average lifecycle. This means that in some cases the authentication string will need to be renewed due to an excessively short validity period.

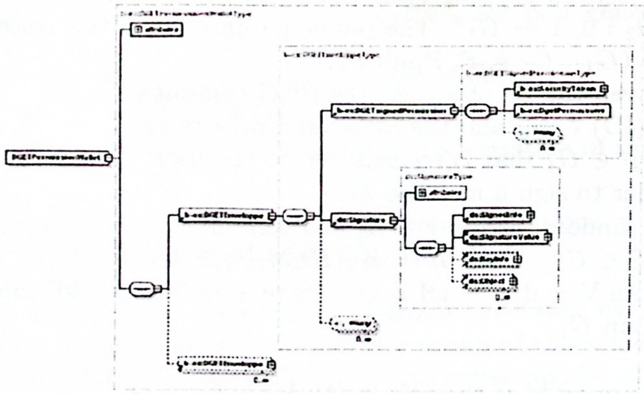

Fig. 3. Entity permissions set Schema

6.3 Policy Repository

Every Nucleus has its own Policy Entity storing its local policies independent of other participants in the grid community. The Policy Entity exposes interfaces to insert, update and delete policies, making the policy administration easier. Policy updates become immediately visible to the authorization decision process. The Policy Entity can be configured to retrieve policies from external sources as well. DGET supports multiple levels of policies governing different aspects of the system. Different policy levels supported by DGET are described below:

Domain Policy Domain policies specify the access control rules defined according to the domain in which the Nucleus is running. Domain policies can be either specified or the Policy Entity can be configured to retrieve a Domain policy from the domain policy repositories.

Nucleus Policy Nucleus policy specifies the policies governing access to system resources and entities. Nucleus policies are specified by Nucleus administrators.

Entity Policy Entity policies decouple access control logic from the Entity application logic and thus updates can be made without changing the Entity code or without redeploying the Entities. Entity policy typically specifies which operations on the Entity are allowed by which users or Entities.

Acting Policy The Acting policy is used for fine-grained permission delegation. The Acting policy follows the least privilege principle and thus allows users to allocate fine grained permissions on the need-to-act basis. The Acting policies controls the level of permissions granted to tasks or requests.

6.4 Authorization and Access Control

Access to scarce system resources and Entities must be controlled and subject to verification of permissions on the resource or perform an operation on the Entity.

Authorization Model DGET uses an association based authorization model. Permissions are organized as authorization profiles. Permissions are granted to profiles and members from the P2P groups, depending on the agreements between the members. Users are required to present credentials to prove their membership with any organization. Based on the membership credentials presented, permissions from the corresponding authorization profiles are granted to the users.

Permission Delegation DGET provides functionality to support fine grained permission delegation following the least privilege principal. Entities can be granted permissions on a need-to-act basis thus avoiding any potential security problem. Users can delegate permissions temporarily to other users who are not members of the organization, or who have rights granted to it. Short-term ad-hoc collaboration scenarios can be supported with this feature. These delegated permissions are attached to the request as the Acting Policy and is evaluated as the part of authorization decision process.

Access Control The Grid is an inherently insecure environment because code is remotely downloaded and executed. Access to shared resources and services must be controlled in order to avoid any misuse. Access control must be exercised at two levels: Nucleus and Entity. The following two paragraphs explain these two aspects of access control:

Nucleus Protection The Nucleus provides the execution environment for Entities. During execution, Entities access the system resources like memory, disk and network etc. It is of utmost importance that access to these resources is controlled. In the absence of such protection mechanisms, some malicious Entities can hijack the Nucleus thus resulting in a Denial of Service (DoS) attack. Nucleus protection is achieved through the Java sandboxing mechanism. DGET uses customized classloaders called GhostLoaders to load ghost classes. GhostLoaders associate appropriate ProtectionDomains based on the authorization profiles.

Entity Protection Besides Nucleus protection, Entities running inside the Nucleus should be protected from misuse as well. Entity owners specify Entity access control policies. These policies are put in place during the deployment. Method invocations on Entities are intercepted and processed through the Authorization Handler. If the method invocation is permitted , the invocation request is processed, otherwise it is rejected.

7 Conclusion

This paper described two main components of the DGET architecture. DGET simplifies the deployment, management and usage of grid systems. DGET provides a dynamic and scalable solution for entity management and security operations that relies on truly decentralized and self-organizing topology. DGET enables resource sharing with the least management overhead and makes grid programming an easier task. DGET provides a flexible interface to adopt any security model. In the future, we plan to incorporate more sophisticated features like fine-grained resource control thus making it feasible to provide Quality of Service (QoS)and support and enforce Service Level Agreements (SLA).

References

1. Adriana Iamnitchi Ian Foster. On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.
2. G. Germoglio N. Andrade, L. Costa and W. Cirne. Peer-to-peer grid computing with the ourgrid community. In *Proceedings of the SBRC 2005 - IV Salão de Ferramentas (23rd Brazilian Symposium on Computer Networks - IV Special Tools Session)*, May 2005.

3. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
4. Domenico Talia and Paolo Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(4):96–95, 2003.
5. Ian Foster. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.
6. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.
7. A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.
8. Adrian Ottewill Benoit Hudzia, M-Tahar Kechadi. Treep: A tree based p2p network architecture. In *IEEE Internatonal Conference on Cluster Computing (Cluster 2005)*, 2005.
9. T.N. Ellahi and M-T. Kechadi. Distributed resource discovery in wide area grid environments. In *The 1st International Workshop on Active and Programmable Grids Architectures and Components APGAC'04, Krakow, Poland*, 2004.
10. T.N. Ellahi B. Hudzia, L. McDermott and T. Kechadi. Entity based peer to peer in data grid environments. In *17th IMACS World Congress, Paris, France*, 2005.
11. T.N. Ellahi B. Hudzia, L. McDermott and T. Kechadi. A java based architecture of p2p-grid middleware. In *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.
12. Time Moses. extensible access control markup language (xacml) version 2.0. In *OASIS Standard*, February 2005.
13. Sun Microsystems Inc. Why are thread.stop, thread.suspend, thread.resume and runtime.runfinalizersonexit deprecated? http://java.sun.com/j2se/1.4.2/docs/guide/misc/ threadprimitivedeprecation.html (visited 16-jan-06).
14. The byte code engineering library (bcel) http://jakarta.apache.org/bcel/ (visited 16-jan-06).
15. Webno Mao. An identity-based non-interactive authentication framework for computational grids. Technical report, Trusted System Laboratory, HP Laboratories, June 2004.
16. H.W. Lim and K.G. Paterson. Identity-based cryptography for grid security. In *Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing (e-Science 2005), Melbourne, Australia*, 2005.
17. Ian F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography*. Cambridge University Press, New York, NY, USA, 1999.
18. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.
19. B. Libert and J. Quisquater. The exact security of an identity based signature and its applications, 2004.