CERIAS Tech Report 2011-29
Accountability for Grid Computing Systems
by Wonjun Lee
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Wonjun Lee

Entitled
Accountability for Grid Computing Systems

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

ELISA BERTINO
Chair

SAURABH BAGCHI

ANNA C. SQUICCIARINI

ARIF GHAFOOR

NINGHUI LI

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): ELISA BERTINO

Approved by: M. R. Melloch 07-18-2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 6/09)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
Accountability for Grid Computing Systems

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Wonjun Lee
Printed Name and Signature of Candidate

07-18-2011
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

ACCOUNTABILITY FOR GRID COMPUTING SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Wonjun Lee

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2011

Purdue University

West Lafayette, Indiana

UMI Number: 3481071

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 3481071
Copyright 2011 by ProQuest LLC.
All rights reserved. This edition of the work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Dedicated to my God, my Lord, Jesus Christ, and Holy Spirit.

ACKNOWLEDGEMENTS

Special thanks to my wife, Eunsung Choi for her love and support. Without her prayer and support, I could not have finished the thesis. I am praying for her complete recovery and believe that God will help her. My mother and parents-in-law gave a lot of encouragement for continuing this work to completion. I thank them.

I deeply thank my advisor Elisa Bertino for her research direction, challenge, and guidance. I also thank Professor Anna Squicciarini for her comments and advising concerning projects and papers.

TABLE OF CONTENTS

Page

LIST OF TABLES ... vii
LIST OF FIGURES ... viii
LIST OF GRAPHS ... x
ABSTRACT ... xi
1. INTRODUCTION ... 1
1.1 Requirements for Accountability Mechanism for Grids ... 2
1.2 Contributions ... 4
1.3 Background ... 5
1.3.1 Grid Computing ... 5
1.3.2 Grid Job Scheduler ... 9
1.3.3 Authentication and Authorization Infrastructure ... 10
2. ACCOUNTABILITY DATA, AGENTS, AND POLICIES ... 11
2.1 Accountability Agents ... 11
2.1.1 Accountability Data ... 11
2.1.2 Locations of Accountability Agents ... 12
2.2 Two Strategies to Collect Accountability Data ... 14
2.2.1 Job-flow Based and Grid Node Based Approaches ... 14
2.2.2 Combination of Two Approaches ... 16
2.3 Log Sharing Mechanism ... 17
2.3.1 Job-graph with Cover-records ... 17
2.3.2 Log Sharing Mechanism in Multiple Domains ... 20
2.4 Guaranteeing Privacy and Non-repudiation ... 22
2.5 Accountability Policy Specification ... 25
2.5.1 Actions' Representation ... 25
2.5.2 Accountability Policies ... 31
3. PROFILE-BASED SELECTION OF ACCOUNTABILITY POLICIES ... 36
3.1 Profile Matcher ... 38

Page

3.2 Accountability Matcher 43

4. VULNERABILITIES IN GRID COMPUTING SYSTEMS 50

4.1 Vulnerabilities of the Connectivity Layer 50
4.2 Vulnerabilities of the Resource Layer 53
4.3 Vulnerabilities of the Collective Layer 55
4.4 Vulnerabilities of the Application Layer 57

5. DETECTION AND PROTECTION AGAINST DISTRIBUTED DENIAL OF SERVICE ATTACKS 62

5.1 Distributed Denial of Service Attacks Involving the Grid 62
5.1.1 Attacks to a Server Located Inside the Grid 63
5.1.2 Attacks to a Server Located Outside the Grid 64
5.2 Tasks of Accountability Agents for DDoS Attacks 64
5.3 Detection Strategies 68
5.3.1 Detection at the Victim Node 68
5.3.2 Detection at the Source Node 71

6. IMPLEMENTATION AND EXPERIMENTAL EVALUATION 74

6.1 Implementations of Agents 74
6.2 Configuration of Experiments in Emulab Test-bed 77
6.3 Experiments 78
6.3.1 Scalability with respect to the number of Computing Nodes 78
6.3.2 Scalability with respect to the number of Resource Providers 81
6.3.3 Scalability across Multiple Domains 84
6.3.4 Scalability with respect to the Data Volume 85
6.3.5 shared policies vs local policies Evaluation 87
6.3.6 Detection and Protection from DDoS Attacks from the Victim-End with Time-Window 1 88
6.3.7 Detection and Protection from DDoS Attacks from the Victim-End with Time-Window 2 91
6.3.8 Detection and Protection from DDoS Attacks from the Source-End 93
6.3.9 False Positive Detection with Two Types of Threshold 95
6.4 An Environment of Accountability Data Queries 97
6.4.1 User Interface and Architecture 97
6.4.2 Querying a Job-graph When Attacks are Detected 99
6.4.3 Querying Accountability Data 102

7. RELATED WORK 107

Page
8. CONCLUSIONS.. 111

LIST OF REFERENCES.. 113

APPENDIX.. 119

VITA.. 121

LIST OF TABLES

Table Page

1.1 Grid Services at Each Layer for Multidisciplinary Job 8

2.1 Symbols Used in the Specification of Actions 24

2.2 Action Specification.. 29

3.1 Definitions of Terminologies Used by Matchers... 44

3.2 Supported Accountability 48

5.1 Classification of Alarms 67

5.2 Collection of Handles and Job-id.. 70

6.1 Data for H1 and H2 from Table 5.2 in Chapter 5 95

LIST OF FIGURES

Figure Page

1.1 Layer Grid Architecture 6

1.2 An Example of Running a Multidisciplinary Job in Multiple Grids 7

2.1 Architecture of the Accountability System 13

2.2 Combination of Two Approaches 16

2.3 An example of Job-graph 18

2.4 Views of a Job-graph. The Circled Portions Denote Different Views 19

2.5 Cover-records for Job-graph of Figure 2.3 When a Job is Submitted 21

2.6 SAML Assertion Containing Handle 22

2.7 Job Contract Publication Process 24

2.8 Job Flow and Corresponding Accountability Data 25

2.9 Abstract Representation of Local Policy and Shared Policy 32

2.10 Accountability Grammar in BNF 34

3.1 Examples of Policy Conflict 36

3.2 The Lifecycle of the Accountability Policies 37

3.3 Example of Profiles for a Job 39

3.4 Example of Risk Factor and Significance Factor 40

3.5 Job-graph Due to Insufficient Accountability 42

3.6 Flow Chart Diagram for Selecting Shared Accountability Policy 46

3.7 Cases of Comparisons with Shared Accountability 49

4.1 Attack Scenario by MD5 Collision 51

Figure Page

4.2 An Example of SQL Injection Attacks in Grid Web-Services 58

4.3 Examples of XML Attack by Using CDATA.. 60

5.1 Distributed Attacks on a Server Located Inside the Grid 63

5.2 Distributed Attacks on a Server Located Outside the Grid.. 64

5.3 Steps for Issuing Alarms 66

6.1 One Use-case of Job Submission... 77

6.2 Job Submission to Multiple Compute Nodes.. 79

6.3 Job Submission Across Multiple RPs 81

6.4 An Example of the Inconsistency in the Handle for Jobs Forwarded through Multiple RPs 81

6.5 An Example of the Handle Consistency for Jobs Forwarded Across Multiple RPs... 81

6.6 Topology for Experiment 6.3.3 84

6.7 Main Screen of the Accountability Grid Computing Portal Before and After Login. 98

6.8 A Cover-record for a Normal Job 99

6.9 An Experiment of Job-Relation (LIST) 100

6.10 A Grid Topology from Emulab.. 101

6.11 An Example of Job-graph 103

6.12 Initial Screen for Querying Accountability Data 104

6.13 Selecting Options for Querying Accountability Records 105

6.14 Result of the Query 106

LIST OF GRAPHS

Graph Page

6.1 Overall Response Time of Job Completion for Different Number of Nodes 79

6.2 Overall Response Time of Job Completion for Different Execution Time 80

6.3 Response Time for Different Number of RPs.. 83

6.4 Average Response Time for Multiple Job Submissions 84

6.5 Data Volume for Different Policies 86

6.6 Comparison of Policy Process Time for Shared and Local 87

6.7 Search Time for Policy Elements 87

6.8 Normal Job's Wait Time for Different Time Windows... 89

6.9 Detection and Recovery Time When the Attacks are Completely Launched for Different Time Windows 92

6.10 Detection and Recovery Time for Different Attack Durations................................ 94

6.11 Probability Distributed for Normal Submissions with Two Different Types of Entropy 96

ABSTRACT

Lee, Wonjun. Ph.D., Purdue University, August 2011. Accountability for Grid Computing Systems. Major Professor: Elisa Bertino.

Accountability is an important security property of distributed systems. It assures that every action executed in the system can be traced back to some entity. Accountability is even more crucial for assuring the safety and security in grid computing systems. Grid computing systems provide a vast amount of computing resources such as computing power, data storage, and network bandwidth. However, to date no comprehensive approach to accountability exists for the increasingly complex grid environments, wherein the number of users and the types of resources are large, diverse, and heterogeneous. Our work addresses this inadequacy by developing a comprehensive accountability system driven by policies and supported by accountability agents. In this thesis, we first discuss the key elements of our accountability framework and types of accountability data obtained in two strategies. We introduce accountability policy that specifies which data to collect and when to collect them, and more importantly how to coordinate data collection among different administrative domains. We then show that the proposed strategies can be realized upon accountability policy by sharing it among accountability agents.

In order to guarantee full accountability without conflicts when the policy is shared, the enforced accountability policies should be adapted based on the different risk levels of jobs and the different significance levels of a node. The support of flexible policies helps protect grid computing systems against malicious jobs, by increasing the level of accountability. To enable support of adaptable accountability policies, we propose a profile-based policy selection mechanism. This mechanism uses profiles of

each job and node and considers node's capability to determine the level of accountability policy for the job and the node. We show how this mechanism can adapt the accountability policies, while at the same time achieving at least a minimum level of accountability.

Accountability data collected by the accountability agents according to the flexible accountability policies provides a basis for analyzing resource usage and finding bottlenecks and detecting security breaches. Additionally, data concerning user activities and actions enables mechanisms for timely identifying malicious users of faulty nodes and helping administrators to take proper defensive actions. In this thesis, we show how accountability data can be used to detect distributed denial of service attacks performed by exploiting resources made available by grid systems to suspend mission-critical websites or the grid itself and then to protect systems from these attacks. We present two approaches for protecting against attacks targeting sites outside or inside the grid.

In the thesis, we also describe a fully operational implementation of our accountability system and report the results from extensive experimental evaluations of it. Our experiments, carried out using the Emulab [1] test-bed, demonstrate that the implemented system is efficient and scalable for grid systems consisting of large numbers of resources and users. In addition, our experiments show that our system efficiently detects the distributed denial of service attacks and is effective in protecting the normal jobs.

1. INTRODUCTION

Grid systems [2] integrate computational and data resources located at numerous facilities, which users can access directly at resource providers or through science gateways. The dynamic and multi-organizational nature of grid computing systems requires effective and efficient accountability systems able to scale for large number of users and resources. The availability of detailed and complete accountability data about users' accesses to grid resources and job executions is crucial for both the grid administrators and the overall grid community. Such data provides a basis for analyzing resource usage, and finding bottlenecks and detecting security breaches. It can also help in managing peer-reviewed resource allocations, authorization, resource accounting and other coordinated services. Additionally, data concerning user activities and actions enables mechanisms for timely identifying malicious users of faulty nodes and helping administrators to take proper defensive actions. Note that limiting the damages in case of security incidents is a major requirement as the consequences of attacks exploiting high performance computing are potentially devastating [3][4].

In current grid systems, OS accounting and monitoring mechanisms [5][6] provide methods to associate CPU, memory, network, and disk usage with specific processes and local principals. A significant amount of information about processes can also be extracted from operating systems, for example from the */proc* file system in Linux. However, current mechanisms are not sufficient to support full accountability because they do not allow resource usage in the system to be monitored at various levels of aggregation. Moreover, in systems in which jobs are decomposed and merged, sometimes unpredictably, mechanisms are required to monitor activities performed across multiple domains.

The design of accountability mechanisms is particularly challenging due to the heterogeneous nature of grid software and system components. To date, there is no grid computing system that addresses multi-domain accountability as part of its information assurance component. Our research addresses this critical inadequacy by developing an accountability system characterized by a rich and flexible language for the specification of accountability policies and an agent-based system to enforce the policies expressed in this language.

1.1. Requirements for Accountability Mechanism for Grids

The design of accountability mechanisms is a complex task that has to meet several requirements in order to overcome the limitations of current logging systems developed for monitoring users' activities and jobs execution. Based on our hands-on experience in the context of the TeraGrid system [7], we have identified several crucial requirements for a suitable accountability mechanism for grids:

Decentralization. It implies the distribution of the accountability tasks across grid nodes. Because of the distributed nature of grid systems, accountability cannot be addressed in a single location, but it must involve all the nodes where a job is processed. This requirement also calls for a harmonic and consistent view of the logging information that follows from the job flow across nodes.

Scalability. Scalability in our context has two dimensions: users and nodes. Today, grids have become widely accessible to large user communities because of the availability of web-based portals. Such communities have an impact on the number of job requests that are typically submitted to grids. Additionally the size of grid systems is increasing because more and more organizations are interested in sharing resources across grids. It is important to devise solutions that scale, and thus work properly for grids of almost any size, from the ones consisting of few nodes to large infrastructures with thousands of nodes.

Flexibility. A rich collection of information should be collected and efficiently stored for later use and analysis, ranging from user authorization data to resource usage information. The system should be able to combine heterogeneous accountability

information as needed. It is however important to identify and select only the data relevant for accountability, as it is not feasible to simply collect all the potentially useful data. The identification of the type of data to collect including information about the users, jobs, and nodes should be specified by using a high-level policy language to simplify administration tasks.

Minimum Impact. The accountability tools must be lightweight and must not interfere with the ordinary computation and activities performed by the grid nodes.

Administration Autonomy. In the design of the system, non-technical barriers such as the coexistence of multiple administrative domains in the same grid system should be taken into account. Note that this requirement is challenging, because of the difficulty to exactly predict how grid administrators will manage their resources. For instance, it is hard to predict to what extent different administrative domains will trust each other in sharing local information with other sites. A good design should thus preserve the autonomy of grid sites, and limit as much as possible the level of collaboration required for the sharing of accountability data.

Integration with Digital Identity Management and Access Control Systems. Because actions executed in a grid system ultimately have to be traced back to real users, it is important that the accountability system be integrated with the system in place for managing user identities. In addition, in order to connect all accountability information related to the same job, the accountability system must be aware of how users are identified across different domains. Integration with access control systems is important in order to determine which access control policies and/or which credentials permitted access to a given user, when an unintended access by this user occurs. The administrators may obtain information useful for revising the access control policies in place and the credentials required to gain access to the grid resources by analyzing accountability data concerning access control decisions.

Detection and Protection from Distributed Attacks. The scalable nature and the complex architectures of grids suffer of several vulnerabilities, since grids were designed with no security in mind. By exploiting its existing vulnerabilities, malicious parties can take advantage of resources made available by grid systems to attack mission critical websites

or the grids directly. Since the attacks we consider here are caused by grid resource and lead to serious consequences, the accountability system should be able to detect the signs of such distributed attacks by monitoring jobs and resources usage and simultaneously protect the grid system.

1.2. Contributions

In this thesis we propose a comprehensive approach addressing the identified requirements based on a layered architecture for end-to-end accountability. We introduce the concept of *accountability agents* or *agents* for short, which are entities in charge of collecting accountability data and monitoring submitted jobs and their users. We develop a simple yet effective language to specify the relevant accountability data according to some policies, referred to as *accountability policies*. The accountability policies specify which data to collect and when to collect them, and more importantly how to coordinate data collection among different administrative domains. Our architecture supports different types of accountability policies. One of them is the *shared policy* that specifies the elements required from an agent in order to obtain a unified form of job execution record. Agents should keep a consistent shared policy in order to guarantee full accountability. However, if elements of the data to be sent from a node to another are missing or different from the ones required by the policy, a conflict may occur. A conflict indicates the inability of a node to comply with the policy shared by nodes. In addition because of different node capabilities and limited amounts of resources available for collection of accountability data, it should be possible to have different shared policy for each job and node. In order to address such conflicts and yet achieve a flexible accountability system, we propose a profile-based policy selection mechanism. Under this approach, the best accountability policy is chosen based on the attributes of jobs and grid nodes, and the capability of each node to collect accountability data. The selected policy preserves the minimum level of accountability and approximates the requirements of the shared policy.

Accountability data collected in a distributed manner according to these dynamic accountability policies provides information about job's trace and its origin and is

analyzed for runtime anomalies. This real-time based diagnostic approach through data analysis plays an important role in detecting the source of malicious activities and identifying the misbehaving parties via a distributed query during forensic analysis. To show how the accountability system can be used for such purposes, we propose an accountability-based mechanism for protection from Distributed Denial of Service (DDoS) attacks conducted by using the resources of a grid computing system. A DDoS attack makes a computer resource unavailable to legitimate users. We discuss two different kinds of DDoS attacks that could exploit grids, and the detection strategies for each kind. Accountability agents leverage information about jobs and resources consumption to quickly detect suspicious patterns that could be symptoms of a DDoS attack. Through a distributed notification protocol, all agents are informed of ongoing attacks and are able to timely react to protect the jobs of legitimate users.

We implemented the prototype of the accountability system on an emulated grid test-bed, which consists of a hundred nodes. Our experiment show that the implemented system is efficient and effective in terms of scalability and protection against DDoS attacks.

1.3. Background

We begin with the overview of the key components of a grid systems followed by an illustration of the authentication and authorization protocols typically adopted in grid systems. We assume that authorization protocols are based on the well-known attribute-based access control model, which is a widely used model for open distributed systems today. Examples of such protocols are those developed as part of the GridShib [8] initiative.

1.3.1. Grid Computing

Grid computing or computational grid is the application of multiple computing resources to a single problem at the same time. A complex scientific or technical problem typically requires a large number of computer CPU cycles and/or a large amount of data. Grids enable sharing and aggregating a wide variety of resources such as supercomputers,

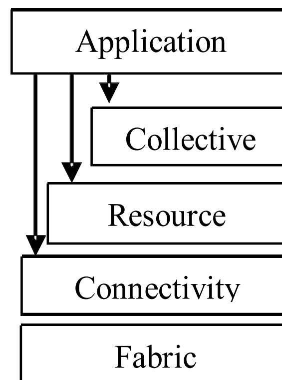


Figure 1.1 Layer Grid Architecture

storage systems, data sources that are geographically distributed and owned by different organizations to solve a large scale computational problems in science, engineering, and commercial enterprises. Grid computing is a form of distributed computing where many networked computers compose a set of clusters [9] to perform very large tasks.

The grid architecture can be viewed as having several “layers” [10] (see Figure 1.1): The grid *Fabric layer* provides shared resources such as computational resources, storage systems, catalogs, network resources, and sensors to which the access is mediated by grid protocols. A “resource” can be defined as logical entity such as a distributed file system, computer cluster, or distributed computer pool. Grid-specific network transactions require communication and authentication protocols. The *Connectivity layer* implements and makes available these protocols. The communication protocols enable exchanging of data between Fabric layer resources, while authentication protocols built on communication services support secure communication with the verification of users' identity and resources. Communication functions include transport, routing, and naming. Authentication solutions have following characteristics: Single-Sign-On (SSO), delegation, integration with various local security solutions, and user-based trust relationships. Grid Security Infrastructure (GSI) [11] is one of services in this layer. The *Resource layer* supports protocols for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. The resource layer protocols are only concerned with individual resources. The Grid Resource Access and Management (GRAM) [12] and GridFTP [13] are examples of protocols in

resource layer. The *Collective layer* supports protocols for interactions across collections of multiple resources such as Condor-G [14] for co-allocating and scheduling services and MPICH [15] for programming systems enabled by grid, while the Resource layer is focused on interactions with a single resource. The final layer, aka *Application layer,* comprises the user applications. This layer provides end-users with access to the underlying resources in the form of command line tools, desktop applications, or web-based interfaces.

The following example shows a usage scenario that commonly occurs in practice and corresponding grid services.

Example 1. Pete, a participant of the open science grid (Virtual Organization, VO) which links shared resources, performs a multidisciplinary simulation, *nwFluid_linux that* uses programs and data located at multiple locations as Figure 1.2. Even though Pete is affiliated with Purdue University, he can run program A at A-state University, and B at B-state University using input data from C-state University.

Based on such scenario we may illustrate how the grid architecture works. Table 1.1 shows the services at each grid layer that might be used to implement the multidisciplinary simulation application in our scenario.

Key components in the grid computing are represented by grid nodes. A grid node is any machine or cluster of machines that processes a job or portion of it. On a typical

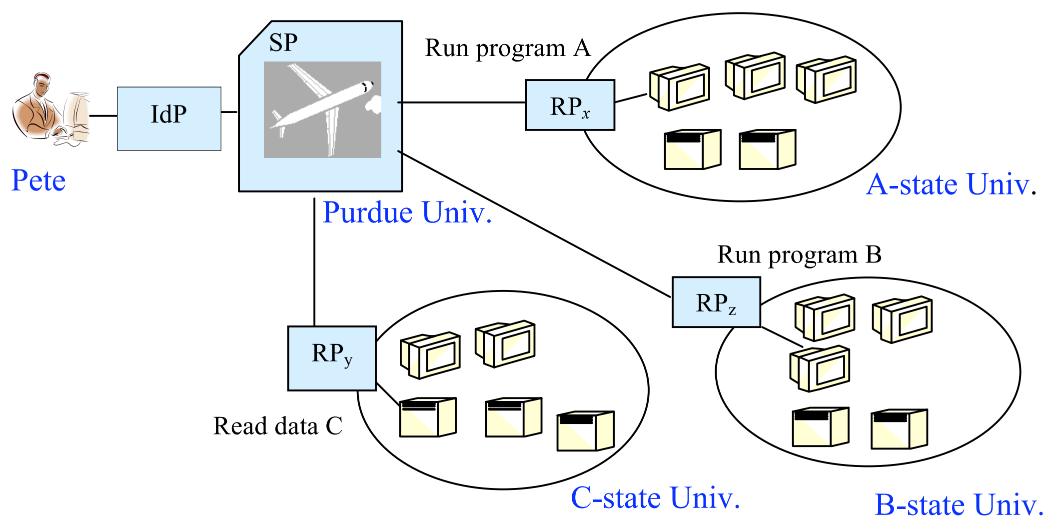


Figure 1.2 An example of running a multidisciplinary job in multiple grids

grid, such as the NSF TeraGrid and Open Science Grid (OSG), each contributing organization, aka Resource Provider (RP), makes available to the grid various kinds of resources, such as computational and visualization resources, datasets, storage, and applications. RPs are typically composed of multiple machines, which may be organized into high-performance computing clusters (HPC). HPC clusters are sets of tightly connected computing machines typically deployed to increase performance by supporting parallel execution of different parts of a job across several nodes in the cluster. In the case of computing resources, each RP typically makes available one or more clusters. Service Providers (SPs) provide specialized services at the application layer, and perform functions such as account management, certificate management and user support. In general SPs make available those services as web services that can be invoked through web portals, also known as *science gateways* such as [16][17].

There are two ways by which grid users gain access to grid resources. The traditional paradigm is for a user to log in to the RP site on its grid entry nodes, which we

Table 1.1 Grid Services at Each Layer for the Example Scenario from Figure 1.3

Layer	*Grid services*	*Remarks*
Application	Multidisciplinary simulation	User applications
Collective	Querying an information to determine availability of computers, storage, and the location of input data	Brokering services for resource discovery; Membership and policy services for keeping track of who is allowed to access resources
Resource	Submitting request to appropriate computer, storage to start computations, and move data; Monitoring the progress of computations and data transfer	Running the same program on different computer systems depends on resource-layer protocol
Connectivity	Obtaining required authentication credentials to submit a job	Must be implemented everywhere, and relatively small; Core protocols are Communication (IP, TCP/UP) and authentication (SSO, delegation)
Fabric	Storage systems, computers, networks, code repositories, catalogs	Physical devices or resources that grid user want to share and access

call *grid entry point,* and submit applications directly to grid nodes using grid middleware commands. With science gateway portals, a researcher can become a user of the portal and, after authenticating at the portal, request services through the portal, which in turn executes the application requested on local or remote grid resources on behalf of the user. In this case, the access to grid resources is transparent to the user, making it possible for a much broader community to utilize high performance grid resources.

It is critical to have common grid infrastructure software in order to construct a grid computing environment. Globus Toolkit [18] is the de-facto standard for grid world. By providing a PKI-based certificate solution for security, it contributed to enable cross-institutional resource access control. As important functions, it provides protocol and services for job submission and resource discovery.

1.3.2. Grid Job Scheduler

Many scientific and engineering applications need to carry large-scale computations. Efficient parallel implementations (e.g., using MPI library – Message Passing Interface [19]) allowed them to run such computational tasks on multiple nodes simultaneously. As a result, a grid job is often a composite of sub-jobs that are scheduled onto available computing nodes by the grid scheduler at the RP.

Portable Batch System (PBS) [20] is a widely used software application that performs job scheduling. The primary job of PBS is to allocate computational tasks among available computing nodes. PBS is a scheduler mechanism supported by GRAM, a component of the Globus Toolkit. As another framework, Condor [14] is prevalently used for job scheduling and supported by GRAM. Condor is a high-throughput computing software framework for coarse-grained distributed parallelization for computationally intensive tasks. It can be used to manage workload on a dedicated cluster of computers or send out work to idle desktop computers. Condor supports the standard MPI and PVM (Parallel Virtual Machine [21]) for the world of parallel jobs.

Like PBS and Condor, most job schedulers run on the dedicated clusters. Each cluster has a head node and several compute nodes (or called worker nodes). The Head Node (HN) is responsible for scheduling jobs based on the resource state as reported by

the compute or worker nodes (CN/WN), the priority of the job owner on the resource. In the case of computing resources, one RP typically makes available one or more clusters.

1.3.3. Authentication and Authorization Infrastructure

As accountability has strong ties with authentication and authorization, it is important to clarify the underlying mechanisms adopted for these crucial security functions. Our accountability system is integrated with the federated approaches used for managing grid user identities, as developed by the GridShib [8] or ShibGrid [22] project. Such approaches do not require cumbersome static pre-registration phases typical of conventional access methods for grid users.

Each user in a Shibboleth [23]-enabled grid system is associated with a unique Identity Provider (IdP), which is the user's home organization. The IdP manages the user's registration, by issuing an X.509 [24] certificate to the user, or if the authentication is not PKI-based, by assigning a login name which is unique within the home organization. The IdP also manages user identity attributes and issues temporary identifiers, referred to as *handles* that are used by the IdP to provide user's attributes to relying parties requesting these attributes. By exploiting the GridShib SAML [25] tool, handles can be embedded in X.509 certificates and pushed to the RP when the user submits a job request. This approach allows the RP to immediately verify the users' attributes and decide whether or not to grant access. The use of handles protects the privacy of user identification from the RP, because RP does not need to know them. However, it makes harder to associate the identity of the user with the submitted job upon its completion, as the actual identity is not included in the temporary handles for privacy purposes. How to achieve accountability when handles are used will be discussed at section 2.2.4.

2. ACCOUNTABILITY DATA, AGENTS, AND POLICIES

To hold individual users accountable for their activities in grid systems, appropriate information should be collected. We have devised two basic approaches to gather such accountability data; *job-flow based, grid-node based*. Data obtained according to those two approaches are then combined to get more detailed aggregate accountability data. In what follows, we begin with describing the type of relevant data collected for accountability, followed by the two basic approaches. Section 2.1 introduces the notion of *accountability agent*. We propose two strategies to collect accountability data by accountability agents in Section 2.2 followed by the log sharing mechanism in Section 2.3. Section 2.4 shows a mechanism of non-repudiation required in accountable grid computing systems. We then introduce a policy language in Section 2.5.

2.1. Accountability Agents

Accountability agent is the entity that collects and processes accountability data based on the two strategies that we proposed. Since the main purpose of accountability agents is to collect data, it is important to identify the type of data that is relevant for accountability.

2.1.1. Accountability Data

Specifically, the following data types are of interest for accountability purpose:
Access control data. Such data is extracted from software at the application layer. It refers to the authentication tokens used by users to access the Grid, the type of credentials (or handles) requested for obtaining authorization, and the corresponding access control

policies utilized, if any[1]. Because access control determines which jobs are executed on the grid, monitoring access control decisions by recording all information related to such decisions is crucial to determine if and why wrong access control decisions have been made and thus take proper corrective actions.

Job-related data. This data is associated with the job and its execution, and is extracted from components of the middleware layer. It includes information such as the number of sub jobs, the machines where the jobs are hosted, the resource (computational and/or storage) consumption for processing the job, the process id, the SP id. Additionally, information related to the protected files accessed by the job can be collected.

Resource oriented data. This data includes the entire information specific to the machine where grid computations are executed, such as resource usage, frequency, number of CPU cycles.

Agents employ different techniques for data collection, according to the type of data they extract[2]. For example, accountability information can be extracted from text logs typical of job schedulers or by intercepting information logged at user portals. Such information tracks users' requests and authorizations about job scheduling.

2.1.2. Locations of Accountability Agents

The functions of agents are twofold. First they monitor resource consumption and/or users' access to the nodes they are associated with. Second, they provide accountability data to other agents. Consequently, to provide a global solution to accountability within the grid and to maximize the benefit of our accountability mechanism, agents must be carefully distributed across the grid nodes.

---

[1] In some grid system access is static and predefined. In those cases, grid mapfiles mapping local accounts

[2] Not all nodes have the same functionality, so different nodes will be mapped onto agents with specific techniques for accountability data extraction

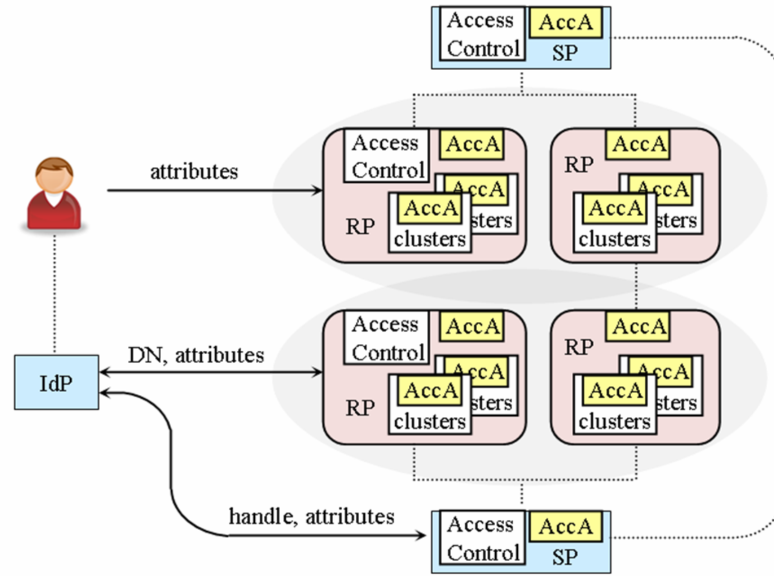


Figure 2.1 Architecture of the Accountability System

Several distribution strategies could be adopted, based on the number of administrative domains, sites and/or distribution of computational nodes. For instance, one could distribute agents so that all administrative domains have a single centralized agent, or agents could be independently distributed using a machine-centric approach and then connected according to the dynamic connections generated by the submitted jobs. We thus identify two main criteria when placing the agents. For each administrative domain we require that there exist at least one agent collecting data for each type of accountability data; and that each possible job flow be monitored by one or more agents from the time of submission until completion regardless of the number of nodes involved and the number of crossed domains.

Based on these criteria we have developed an articulated strategy for agent location. At each RP - corresponding to an administrative domain -, agents are located by layers, as shown in Figure 2.1 (AccA is a shorthand for Accountability Agents). A first layer of agents is located at the entry points of the grid. As discussed, these nodes take authentication and authorization decisions. Moreover, users handles and/or authorization tokens are created at these nodes. Agents can thus record here the policies used to authenticate users and/or to grant the required job request. Agents at this layer are

associated with SP machines and/or RPs offering direct access.

A second layer of agents is located with the schedulers such as Condor-G [14] or PBS. These agents collect information related to the job scheduling strategy, such as the RPs where the job will be processed and, in the event of a job split into multiple sub-jobs, the number and destinations of these sub-jobs.

Finally, a last layer of agents is located at the compute resources. Our design requires at least one agent for each head node. The cluster head node hosts agents because the head node schedules jobs to the compute nodes and has job related information. The head node is responsible for compute nodes, as its main function is to control and monitor compute nodes. Existing monitoring primitives allow head nodes to retrieve aggregate accountability data about compute nodes.

However, such primitives do not neither track how the job is split, nor do they track the resource consumption for each sub-job created. To obtain such fine-grained information and achieve full accountability we require each compute node to have an agent. Data at compute nodes is collected using an accounting tool and sent back to the head node upon request.

2.2. Two Strategies To Collect Accountability Data

2.2.1. Job-flow Based and Grid Node Based Approaches

Agents operate according to two different strategies, namely *job-flow based* and *grid node based*, with emphasis, respectively, on data related to jobs and their flow; and on the sources of specific data types.

Jobs flow from the entry point to the remote grid nodes based on resource availability and job description. As mentioned in Chapter 1, a job that requires a long computation is often split into many sub-jobs to be executed in parallel. Sub-jobs are distributed across different grid nodes, and move from nodes to nodes. Hence, monitoring sub-job transfer is crucial. A possible approach is to employ point-to-point agents, which collect data at each node that the job traverses. We refer to this approach as a *job-flow*

based strategy. Such approach enables tracking a job process throughout its whole life-cycle, from the time when a process is created to the time when its execution is completed. Each agent only controls the node where it is located, because due to the distributed nature of grids it is not practical to collect all information about a job at a centralized location.

The data collected according to the job-flow based approach is of two kinds: *access control data* and *job-related data*.

In order to obtain a complete picture of the grid active jobs and the related resource consumption, the job-flow strategy is complemented with an approach collecting *resource-oriented data*. This approach, referred to as the *grid-node based* strategy, collects data at a given spatial location for all jobs active at such location a given time point. The spatial location may be set at one or multiple nodes [26] of the grid system. With respect to the type of data collected, the grid-node based strategy focuses on collecting resource-related information, which includes entire information specific to the machine where jobs move and computations are executed. The viewpoint can be restricted at data related to a single grid layer.

In the grid-node based strategy data is collected by exploiting appropriate resource monitoring tools, which periodically collect resource usage information at each fixed points. Specifically, if the focus is on the fabric layer, such information includes used CPU cycles, state information such as current load, queue state, memory usages for computational resource. If the focus is on the connectivity layer, since data is exchanged through communication protocols, transport, routing, and naming information for each job can be logged for all the active transactions. Finally, if the focus is on the resource layer, where operations, such as process creation and data access, are performed, process information and/or file names transferred may be of interest. The agents take advantage of such existing information sources -so to meet the minimum impact requirement - without adding new monitoring mechanisms when possible.

2.2.2. Combination of Two Approaches

As accountability data can potentially be used for diverse types of analysis, an approach focusing on one single aspect may be inadequate. Moreover, only the combination of different types of data can provide a solid basis for analyzing the use of the grid and identifying possible misuse of grid resources.

The aforementioned approaches are complementary to each other, and can be used to collect detailed information about the executed job, along with its resource consumption and status progress at each traversed node. For example, if a job misused the resources available at a certain node (by for example accessing protected files), by retrieving its job-id and analyzing resource data collected by the grid-node based method, we can identify the actual principal who submitted the job. Furthermore, we can investigate other possible errors of related jobs, which used the same resources and have been submitted by other principals. Such detailed analysis is possible only by cross correlating the data collected using the job-flow based method.

Figure 2.2 shows a simple example of data sets collected by the two approaches. The first two tables show the names of data and their values extracted by the job-flow strategy. One of jobs, identified as PBS.3839, is submitted to an SP named gk.rcac.purdue.edu. The handle 3f7b3dcf-1674-4ecd-92c8-1544f346baf8 is generated by IdP idp.rcac.purdue.edu. Since the job is a multi-job divided and submitted to RP2 and RP3, the identifiers of sub-jobs, PBS.3839-2 and PBS.3839-3, should be collected as paired with their destinations. Assume that the sub-job, PBS.3839-3, is suspended for some reasons and the accountability policy specifies to log the process identifier that is,

Handle	*3f7b3dcf-1674-4ecd-92c8-1544f346baf8*
Job Id	PBS.3839
{Sub jobs, Dest.}	PBS.3839-2 --> RP2
{Sub jobs, Dest.}	PBS.3839-3 --> RP3
Job-Relation	Graph 1
Date/Time	*2007:02:08:09:48:22*
Checking-Point	Job Queued
Node Id	gk.rcac.purdue.edu
IdP Id	idp.rcac.purdue.edu

SP

Handle	*3f7b3dcf-1674-4ecd-92c8-1544f346baf8*
Job Id	PBS.3839-3
Job-Relation	Graph 3
Process Id	**11325**
Date/Time	*2007:02:08:10:11:19*
Checking-Point	Job Suspended
Node Id	cn5.rcac.purdue.edu

CN

Process Id	7193
Memory Usage (MB)	65
Process Id	**11325**
Memory Usage (MB)	*1910*

Resource: Memory
Host: cn5.rcac.purdue.edu
2007:02:08:10:52:00

CN

Figure 2.2 Combination of Two Approaches

11325, for the job suspended in the compute node, cn5.rcac.purdue.edu, then the agent would find the process id mapped to abnormal memory usage at the resource table located at the same node, that is, cn5.rcac.purdue.edu. Once the sub-job, PBS.3839-3 is identified as one that caused misuse of the memory resource from the second table, PBS.3839-2 assigned at RP2 is identified as a job that may potentially cause bad operations, because sub-jobs may be heavily interdependent. In this example, the information in the resource table is obtained according to grid-node based strategy by fixing the point at Fabric layer, while the first two tables are generated according to job-flow based strategy.

2.3. Log Sharing Mechanism

2.3.1. Job-graph with Cover-records

Many scientific applications require multiple computing nodes and run efficient parallelized implementations. As a result, a grid job is divided into many sub-jobs and scheduled to run many nodes. These nodes may reside in different network domains. In practice, a sub-task of a job is sometimes further divided and executed at other nodes. We call such composite jobs the workflows of sub-jobs. A common approach to model job-flows is to employ a directed graph that directly describes how the sub-jobs of a job are interconnected. We refer to such directed graph, representing the flow of job/sub-jobs, as *job-graph*. The vertices of the job-graph represent grid nodes where jobs are forwarded, scheduled and/or processed. The directed edges denote job movements resulting by the scheduling or/and rescheduling of the job and/or sub-jobs onto another grid node because of parallelization, lack of resources in a node, and so forth. Job-graphs are not always generated in real time base.

Definition 2.1 (Job-graph). Let N be a non-empty set of grid nodes. A job-graph $G = \{N, E\}$ is a directed graph satisfying the following conditions: 1) each node $n \in N$ corresponds to a grid node characterized by indexes i, j, where i denotes the unique node

identifier and *j* the computational units of the nodes[3] 2) each edge $e = (n_i, n'_{i'}) \in E$ denotes a sub-job assignment from the parent node n_i to the child node $n'_{i'}$ 3) there is a unique root node of the graph, that is $\exists n \in N$ *s.t.* $(n', n) \notin E$.

Thus, the number of edges in a graph is the same as the number of job assignments. A same node may have multiple entering edges if the same node is assigned to process two or more sub jobs of the same job, i.e., an overloaded node. Job schedulers typically adopt this approach in case of overloaded nodes or computationally intensive jobs.

The graph in Figure 2.3 illustrates an example of job-graph. Suppose that a multi-job *job1* that comprises two sub-jobs, *job1-a*, *job1-b*, is submitted for execution at service provider *SP1*. A sub-job, *job1-b* that is scheduled at resource provider, *RP2*, is further split onto *job1-b-1* and *job1-b-2* to be run in parallel at compute nodes *CN1* and *CN2*, respectively. The directed edge from *CN2* to *CN1* shows that *job1-b-2* is rescheduled at *CN1* because of, for example, insufficient resource at *CN2* for job *job1-b-2*. If *job1-b-2* is evicted from node *CN1* for the same reason and then rescheduled to *CN2* again, the identifier of *job1-b-3* should be assigned a name, for example, *job1-b-4*, different from already assigned names in order to distinguish activities performed before and after the evictions. Especially when the suspicious operation is repeated making loops between nodes, renaming helps to trace back to the original job by constructing cover-records based on the modified job names.

Once created, the job is a moving object that traverses grid layers to reach multiple nodes, and finally consumes resources in the fabric layer.

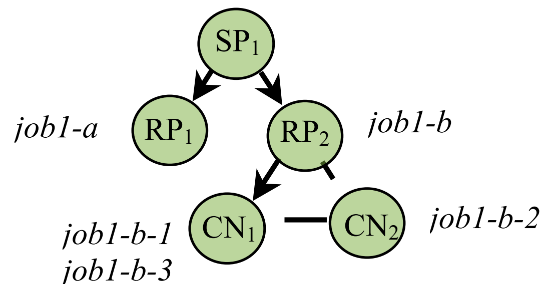


Figure 2.3 An Example of Job-graph

[3] Recall that as specified in the past section a same grid node can have multiple computational units

The main challenge in enforcing the job-flow based strategy is represented by the ability of tracking a job, in presence of complex job scheduling techniques adopted by grid nodes. Our approach to capture provenance information during the execution of a job is based on two key factors: 1) the use of shared policies, and 2) the design of a graph based log sharing mechanism. Specifically, the accountability agents share the *job-relation data* with communicating agents, as specified by the shared policies.

Each agent stores a view of the job-graph defined by Def. 2.1. The view is defined from the perspective of one grid node (controlled by one agent) corresponding to a node in the job-graph. Graph views are defined as follows:

Definition 2.2 (Graph View). Let $G = \{N, E\}$ be a job-graph, defined according to Definition 2.1. A view of a job-graph from a node $n_{z,t}$ is defined as $V = \{N', E'\}$ where:

1. $\exists n_{i,j} \in N'$ s.t. $n_{i,j} = n_{z,t}$;
2. $E' = \{e \mid (e \in E \wedge e = (n_{z,t}, n) \vee e \in E \wedge e = (n, n_{z,t}))\}$;
3. $N = \{n \mid (n \in \mathbf{N} \wedge (e = (n, n_{z,t}) \wedge e \in E') \vee (e = (n, n_{z,t}) \wedge e \in E'))\}$

By definition, each agent has a partial vision of the job path (see Figure 2.4) that includes the immediate predecessor node and the immediate successor node(s) (subject nodes).

Each accountability agent keeps a *cover-record* that keeps track of the job-relation between predecessor and successor nodes.

The cover-record maintains the local graph view as Def. 2.2, along with additional

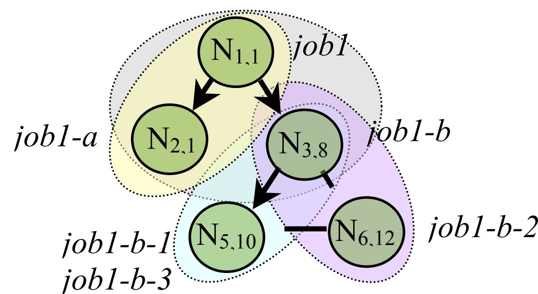


Figure 2.4 Views of a Job-graph. The Circled Portions Denote Different Views

information, such as job-id, handle value for unique identification, and other data as specified by the shared accountability policies (see Section 2.3). Cover-records also maintain a log of the job state, along with the timestamp tracking the job state transition from one state to another. The union of the graph views determined by the job sharing mechanism corresponds to the whole graph, provided that the shared policies support a correct sharing of the information required to connect the shared jobs. The truth of such claim follows from condition 1 of Def. 2.2, which ensures that all nodes in the graph are considered, and from the fact that all the relationships among nodes are captured, as a consequence of condition 2 in Def. 2.2.

2.3.2. Log Sharing Mechanism in Multiple Domains

To implement the job-flow based logging method, the agent in each node follows two rules as discussed in subsection 2.1.2. First, the agent logs a partial path of the job that includes the direct predecessor and direct successor nodes. Second, each agent shares the collected information obtained at its node with other agents based on the stated *accountability policy*. An accountability policy specifies the exact accountability information to be collected, as we will discuss in Section 2.5. This approach, referred to as *graph-based log sharing mechanism*, is highly decentralized, in that no single agent keeps track of the whole job-flow. In other words, the agents maintain complete view of a job as a group, without generating a large amount of overhead at a single node.

Operationally, each agent keeps a cover-record that shows the relation between the job, or a portion of it, in its node (namely, the *subject node*) and the ones allocated at direct predecessor and/or direct successor nodes (*object node*). The cover-record maintains the local view of the whole graph in the job-relation information. The agent generates a cover-record when specific changes in a job state occur. Examples of such states as supported by Globus Toolkit [18] are *pending, active, suspended, completed*. For instance, when the job moves into a pending state, the corresponding cover-records are generated in each node as shown in Figure 2.5.

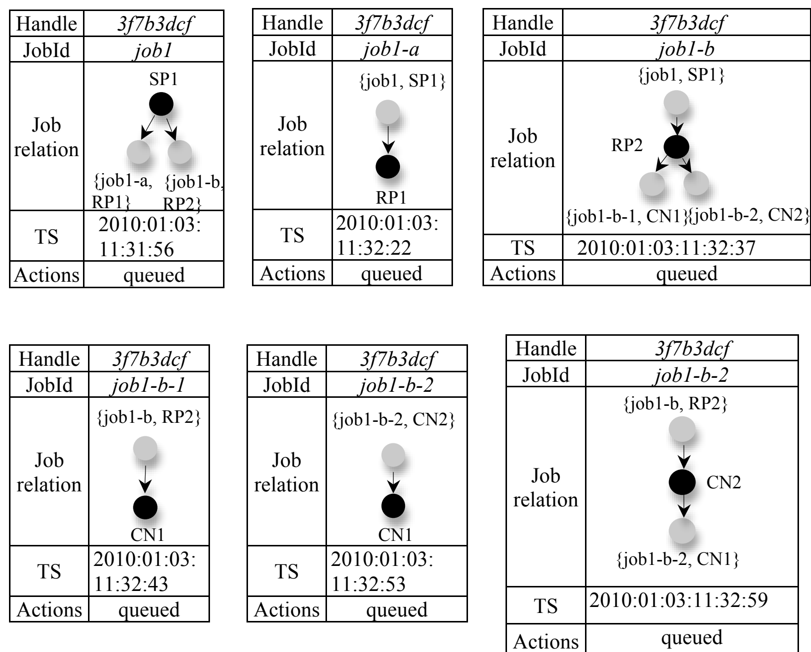


Figure 2.5 Cover-records for Job-graph of Figure 2.3 When a Job Is Submitted. – Clockwise from Upper Left, (a), (b), (c), (d), (e), and (f)

Merging the local views represented by cover-records results in a whole graph. The job-graph is a single rooted graph structure, and the root node is the job entry point (i.e., the service provider or gateway where the job is submitted). The job entry point is the natural root since it stores at first the job record. When the whole graph structure is to be completed for a split or forwarded job, the agents relay job information to their father nodes, so that the agent at the root node is able to assemble the whole job information of a job-graph. The agent at each node is responsible for forwarding the job and/or resource information collected locally to the predecessor node. When the node does not have a child node in the cover-record (e.g., nodes in Figure 2.5-(b), (d), (e)), the agent at the node just sends the resource related data. However if there are successor nodes for the subject node, like in the examples in Figure 2.5-(c), (f), the agent is in charge of collecting data that its successor nodes received or collected. Finally the agent at the root node, SP1 in Figure 2.5-(a), is able to collect all accountability data.

2.4. Guaranteeing privacy and non-repudiation

One of Internet2 working groups [23] for Shibboleth has implemented two methods for implementing handles in the transactions. These are the *SharedMemoryShibHandle* [27] and the *CryptoShibHandle*. Shibboleth Single Sign On (SSO) Service generates the *handle*, which is opaque, transient identifier associated with the authenticated user, and then stores the handle and local user name corresponding to that handle at cache memory. This handle is then used to request all available attributes for the user referred to by this handle to the user's home organization. To request a user's attributes the Shibboleth daemon at the SP sends the same handle it received from the user. Between the IdP and SP, the following SAML [25] authentication assertion which contains the *NameIdentifier* [28] is used. In a SAML, the IdP creates a *NameIdentifier* and embeds it in an authentication assertion; SP include the *NameIdentifier* in the SAML assertion to request to IdP.

With the handle received from the SP, Attribute Authority (AA) at the IdP verifies whether the handle is recently generated by the SSO Service and to which user it refers. In general, the user's actual identity is hidden outside of the home organization by explicitly referring to this handle. The randomness of handles is good for privacy since neither SP nor RP can determine real user's identity from handles. Handles are always unique for every individual Shibboleth transaction across SPs. These identifiers have a one-time use semantics since they are kept in a cache and then terminated after being used to search a local user and thus providing the user's attributes to the requesting SP.

Instead of storing the handle at cache memory, with the cryptographic scheme, we are no longer to keep the handle in the cache memory at IdP side. This is another implementation of handle, which is called *CryptoShibHandle* provided by Shibboleth. In *CryptoShibHandle*, the local user name and a random string are encrypted directly into

```
<saml:Subject>
    Format=”urn:mace:shibboleth:1.0:nameIdentifier”
    NameQualifier=https://idp.example.org/shibboleth>
    3f7b3dcf-1674-4ecd-92c8-1544f346baf8
</saml:NameIdentifier>
```

Figure 2.6 SAML Assertion Containing Handle

the handle value. Therefore it does not require keeping a state at the IdP at the expense of using a symmetric key.

In accountability system where all handles should be kept both at the IdP and at the SP to determine the real user identity from the handles, the first method is not appropriate because if the number of users in an organization is high, the handles to be created should be multiplied by the number of job requests and keeping all handles in caches or secondary storages all the time is very expensive.

Although the *CryptoShibHandle* satisfies issues both about privacy and memory usages, it does not satisfy non-repudiation. The term, non-repudiation crypto-technically means a way to provide proof of the integrity and origin of action [29], which can be verified by any third party. It is an important property of accountability to protect against false denial of a certain action. However there is no way to verify that the user is the real identity who is claimed to be when using *CryptoShibHandle*. While the Shibboleth does not specify the authentication method adopted by the IdP, we propose that Public Key Infrastructure (PKI) should be used to give a medium to guarantee non-repudiation to the system. When authenticating a user in PKI, the web server verifies the user at first by sending a nonce, which is a random string, and then by receiving the encrypted nonce by user's private key. Authentication is completed when decrypting the received encrypted nonce with a public key, which is embedded at the certificate, and then matching the original nonce with the decrypted nonce. The IdP should have an additional component to encrypt user's unique identity such as Distinguished Name (DN), email address, etc., both with encrypted and plain nonce for a handle. The handle is as follows.

$$Handle = E_{IdP\text{'s symmetric key}}(\text{local User's identity} + E_{User\text{'s private key}}(nonce) + nonce)$$

If an intentional or unintentional misuse of the resources is detected, the accountability agent requests IdP with the handle for claims. The IdP can identify real user by decrypting the handle with its symmetric key and verify that he/she is who requested the service to the SP, through the user's signature. Since the nonce, which is encrypted by the user's private key, is given in the handle, IdP can easily verify the user's

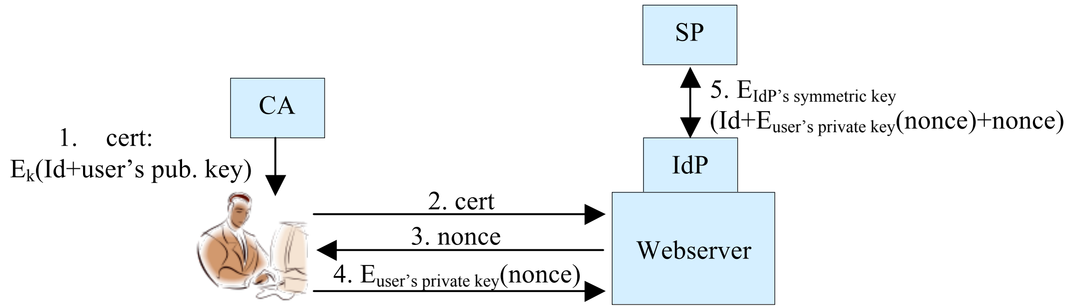


Figure 2.7 Job Contract Publication Process

identity by matching the plain string nonce with the one decrypted by user's public key. If they match, the user is not able to deny his/her past actions.

In this mechanism, IdP does not need to keep nonce information since it is already included at handle, and users' identity information is protected by the IdP's symmetric key against SP/RP. The malicious user cannot deny his/her malicious action conducted as identity in the handle because no one except him/her can know his/her private key that encrypted the nonce. Thus additionally malicious IdP cannot forge the nonce and

Table 2.1 Symbols Used in the Specification of Actions

Name	Symbol	Description
Agents	$\mathcal{A}$	is the set of accountability agents. Each agent $a \in \mathcal{A}$ is uniquely identified by combination of agent id and node location
States	$\mathcal{S}$	is the set of possible states a job can assume
Data	$\mathcal{DS}$	is the set of possible data items to collect. It is partitioned into three subsets, one for each possible data type
Access Control Data	$\mathcal{DS}_{ac}$	subset of $\mathcal{DS}$ that collects access control data
Resource Data	$\mathcal{DS}_{res}$	subset of $\mathcal{DS}$ that collects resource related data
Job Data Set	$\mathcal{DS}_{job}$	subset of $\mathcal{DS}$ that collects job related data
Repository	$\mathcal{R}ep$	denotes the storage repository where accountability data can be located
T	$\mathcal{T}$	Temporal expressions, specified as [30][31]

encrypted nonce by generating another private key to claim non-malicious user since IdP cannot modify user's certificate where the user's public key exists.

2.5. Accountability Policy Specification

Accountability policies specify what to track and when, and more importantly how each agent has to coordinate with other sites' agents. In this section we introduce a high-level representation of such policies. Policies are expressed by actions, capturing the main activities of an agent.

2.5.1. Actions' Representation

We model the agents' basic actions using seven expressions. The main symbols

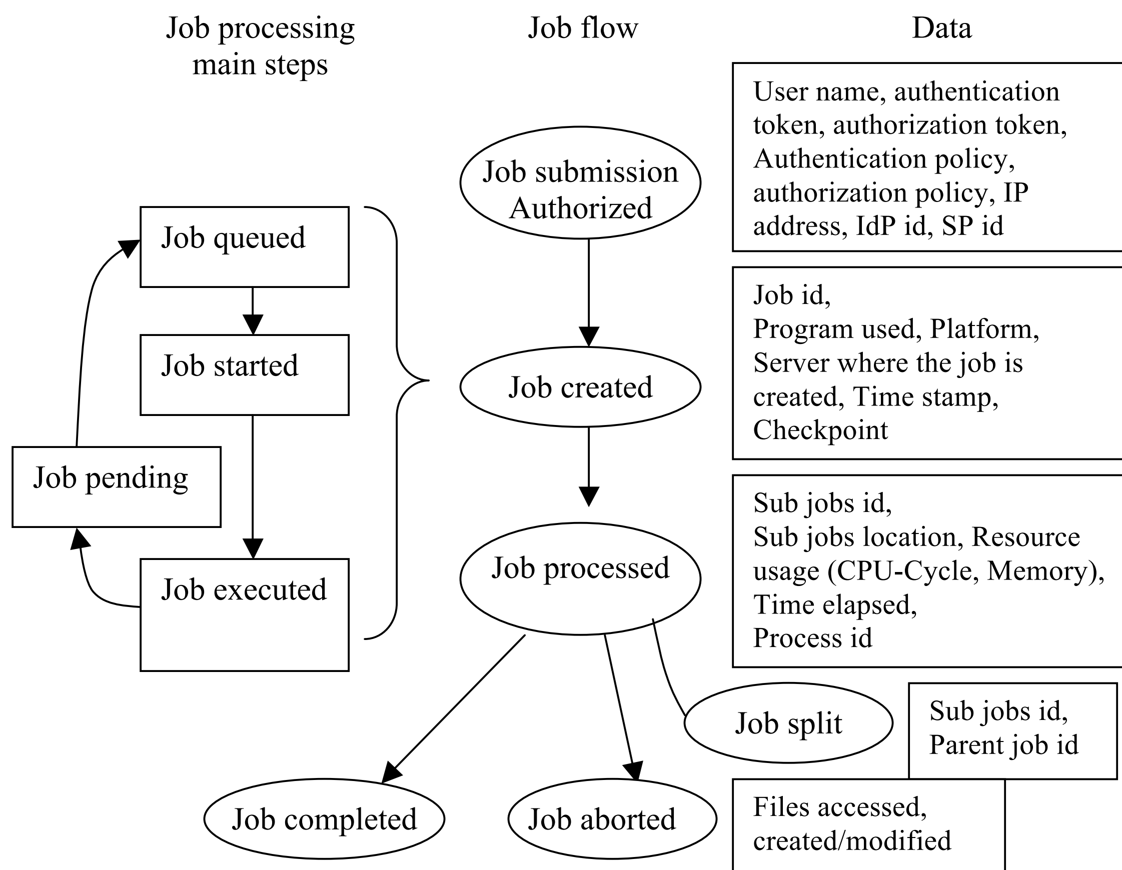


Figure 2.8 Job Flow and Corresponding Accountability Data

used for our expressions are listed in Table 2.1.

Actions describe the agents' operations to be executed, and may or may not relate to jobs. When jobs are involved, agent's actions are also influenced by the job state, which changes over time, from the creation to the completion (successful or abnormal). The set of states that we consider are denoted as $\mathcal{S}$. A generic list of possible states is provided in Figure 2.8[4].

Actions can be of seven different types, and are defined as presented in Table 2.2. Detailed descriptions of each type are as follows.

collect_job_data(x, state, data_set, storage): ***x*** *denotes an agent and takes values from* $\mathcal{A}$ *;* ***state*** *denotes a set of job state and takes values from* $\mathcal{S}$*;* ***data_set*** *denotes the set of data to be collected;* ***storage*** *denotes the data repository* $\mathcal{Rep}$ *where the collected data have to be stored.*

This type of action specifies the information that agent in node has to collect for all jobs locally processed. Note that the mandatory element to be collected for all job actions is the job-id, which is fundamental for binding the job with its data. The exact data to be possibly retrieved changes according to the state of the job at the time of collection. When several state values are listed in the same action, the semantics is that the action is triggered when the job enters one of these states. Intuitively, some states imply others. For example, if a job is queued, it means that it has been already submitted. However, the action should occur only when the specified state is reached. As shown in Figure 2.8, the state can be expressed at different granularity levels. A coarse grained expression may only consider the executing state of job while a fine-grained one may differ among the various job processing steps. We assume data collection to be an atomic operation with respect to the job state. Agents collect the data available upon job transition from one state to the subsequent one, with the obvious exception for terminal states.

collect_resource_data(x, data_set, time_constraints, storage): x *denotes an agent and takes values from* $\mathcal{A}$*;* ***data_set*** *denotes the set of data to be collected and takes values*

---

[4] Specific transition state diagrams can slightly differ depending on the specific job type, whether it is a computing-intensive job or a long-running one.

from $\mathcal{DS}_{ac} \cup \mathcal{DS}_{res}$*;* ***time_constraints*** *denotes a temporal expression and takes values in* $\mathcal{T}$*;* ***storage*** *denotes the data repository* $\mathcal{Rep}$ *where the collected data have to be stored.*

The second type of action specifies the information that has to be collected for a resource according to the temporal constraints specified in *time_constraints*; *time_constraints* is a compound temporal expression, specifying both a periodic expression and the retention time, to mandate respectively how often the data needs to be collected and for how long has to be maintained. Periodic and temporal expressions are expressed using formalism proposed in [30][31].

send_job_data(x, agent_job_relation, state, data_set, job_id): x *denotes an agent and takes values from* $\mathcal{A}$*;* ***agent_job_relation*** *denotes agents who will receive values from* $\mathcal{A}$*;* ***state*** *denotes a set of job state and takes values from* $\mathcal{S}$*;* ***data_set*** *denotes the set of data to be collected and takes values from* $\mathcal{DS}_{job}$*;* ***job_id*** *denotes job_id and takes values from* $\mathcal{DS}_{job}$

In order to build a partial view of the job-graph, agents at each node should send a job-relation information to the node to where the job flows.

receive_job_data(x, agent_job_relation, state, data_set, job_id): x *denotes an agent and takes values from* $\mathcal{A}$*;* ***agent_job_relation*** *denotes agents who will send values to* $\mathcal{A}$*;* ***state*** *denotes a set of job state and takes values from* $\mathcal{S}$*:* ***data_set*** *denotes the set of data to be collected and takes values from* $\mathcal{DS}_{job}$*;* ***job_id*** *denotes job_id and takes values from* $\mathcal{DS}_{job}$

In order to build a partial view of the job-graph, agent in each node should receive a job-relation information from the node from where the job flows.

request_job_data(x, agent_job_relation, data_set, job_id): x *denotes an agent and takes values from* $\mathcal{A}$*;* ***agent_job_relation*** *denotes agents to which request will be made by x ;* ***data_set*** *denotes the set of data to be collected and takes values from* $\mathcal{DS}_{job}$*;* ***job_id*** *denotes job_id and takes values from* $\mathcal{DS}_{job}$

The agent of the root node in a job-graph can trace every traversal of the job across the domains as if every grid node exists in the domain local to the root node. To do this, the root node needs to request data to agents located at a job-graph to ask forwarding their collected data. Agents requested this action would repeat requesting job data to successor nodes until all terminal nodes are reached.

forward_job_data(x, requester, data_set$_{combined}$, job_id): x *denotes an agent and takes values from $\mathcal{A}$;* ***requester*** *denotes agents who requested the values;* ***data_set$_{combined}$*** *denotes the set of data to be collected and takes values from $\mathcal{DS}_{job} \cup \mathcal{DS}_{ac} \cup \mathcal{DS}_{res}$;* ***job_id*** *denotes job_id and takes values from $\mathcal{DS}_{job}$*

The agents requested by the root node or the predecessor node for sending data, are responsible for forwarding the collected data to the requester.

combine_job_data(x, agent_job_relation, data_set$_{combined}$, job_id): x *denotes an agent and takes values from $\mathcal{A}$;* ***agent_job_relation*** *denotes agents who will receive values from $\mathcal{A}$;* ***data_set$_{combined}$*** *denotes the set of data to be collected and takes values from $\mathcal{DS}_{job} \cup \mathcal{DS}_{ac} \cup \mathcal{DS}_{res}$;* ***job_id*** *denotes job_id and takes values from $\mathcal{DS}_{job}$*

To build a complete view of the job-graph, agents at the root nodes need to collect overall information of a job. Before forwarding data, an agent combines collected data obtained from successor nodes with the locally collected data.

The following is an example of actions. In our analysis and examples, we consider the two traditional types of high performance computing job; computations and data transfers.

Example 2. The following action specification states that agent AA@Purdue at Purdue University will collect user's handle, job_id, process_id, and time stamp when a job is either transferred or completed.

collect_job_data(AA@Purdue, {Transferred, Completed}, {handle, job_id, process_id, timestamp}, purdue_db)

Next actions specify the collection of the resource data associated with agent AA@Purdue to be executed every week day once an hour. Following we show examples of action specifications for sending job data.

collect_resource_data(AA@Purdue, DATA, Week+{2,...,6}+1h, purdue_db)

DATA:={CPU cycle, memory consumption, network bandwidth}

send_job_data(AA@Purdue, AA@B-State, completed, {}, job_id)

receive_job_data(AA@Purdue, AA@C-State, submitted, timestamp, job_id)

Table 2.2 Action Specification

Action Type	Arguments	Semantics
collect_job_data	*(x, state, data_set, storage)*	agent *x* collects data in *data_set* about job_id, where job_id is a mandatory element in data_set, when job_id reaches a state among the ones appearing in state and stores it at repository, *storage*
collect_resource_data	*(x, data_set, time_constraints, storage)*	agent *x* collects data in *data_set* at repository, *storage* according to the temporal time constraints, *time_constraints*
send_job_data	*(x, agent_job_relation, state, data_set, job_id)*	agent *x* sends data in *data_set* to agents that belong to *agent_job_relation* for a job, *job_id* when the job state turns to *state*
receive_job_data	*(x, agent_job_relation, state, data_set, job_id)*	agent *x* receives data in *data_set* from agents that belong to *agent_job_relation* for a job, *job_id* when the job state turns to *state*
request_job_data	*(x, agent_job_relation, data_set, job_id)*	agent *x* requests data in *data_set* to agents that belong to *agent_ob_relation* for a job, *job_id*
forward_job_data	*(x, requester,* $data_set_{combined}$*, job_id)*	agent *x* forwards data in $data_set_{combined}$ to agents for a job, *job_id*
combine_job_data	*(x, agent_job_relation,* $data_set_{combined}$*, job_id)*	agent *x* combines data forwarded from agents that belongs to *agent_job_relation* into $data_set_{combined}$ for a job, *job_id*

Actions can be combined and merged in case they are redundant. In order to check for redundancy, actions need to be expressed in a minimal, also called *canonic,*

form. We say that an action is in a canonic form if it only conveys one value for each possible input parameter, excluding the data_set parameter.

Redundancy is defined as follows. In the definition, *A* denotes an action, and let *A.par* denote the parameter name in *A* and let $A.par_{val}$ denote the corresponding values.

Definition 2.3 (Redundant actions). Let *A* and *A'* be actions in a canonic form. We say that *A* is redundant with respect to *A'* if

- ♦ *A* and *A'* are of the same type;
- ♦ *∃A.par' s.t. A.par = A'.par'*
- ♦ and $A.par_{val}=A'.par'_{val}$ and $A.data_set_{val} \subset A'data_set'_{val'}$.

Example 3. Consider the following canonic actions.

(1) collect_job_data(AA@Purdue, Completed, {handle, job_id, process_id, file_name, timestamp}, purdue_db)

(2) collect_job_data(AA@Purdue, Completed, {job_id, file_name, timestamp}, purdue_db)

Since {job_id, file_name, timestamp} ⊂ {handle, job_id, process_id, file_name, timestamp} (2) is redundant with respect to action (1).

Redundant actions can be eliminated – action (2) of Example 3 is eliminated. A set of non-redundant actions mandates a protocol for agents to execute. We define such protocols as action expressions.

Definition 2.4 (Action expressions). Action expressions (AE) are defined recursively as follows:

- ♦ All actions defined according to the specification of Table 2.2 are action expressions.
- ♦ If A and A' are action expressions, then the set AE={A, A'} is an action expression.

Action expressions do not mandate an execution order. However some of action expressions are meaningful only if executed in a certain sequence. For example, if one type of action expressions is of *forward_job_data*, then the action expression should also contain a *combine_job_data*, i.e. expressed as (*forward_job_data* ⇒ *combine_job_data*). Other examples are as follows (*send_job_data* ⇒ *collect_job_data*); (*request_job_data* ⇒ *send_job_data*).

2.5.2. Accountability Policies

The accountability policies are of two types: *local* and *shared*. The two types are the result of the different strategies that an agent can adopt. We recall that these correspond to the job-flow based and grid-node based strategies. Policies local to a given location capture data as required by the grid-node based approach. By contrast, shared policies apply to the job-flow based approach, and specify which job information has to be sent or received upon job change of state, from an agent.

An abstract representation of the policies is provided in Figure 2.9. The policies are actually encoded using XML [32]. Such encoding is represented in Appendix A.

The local policy shown in Figure 2.9-(a) specifies that the agent's action, COLLECT-RESOURCE-DATA must be executed in order to collect resource data when the job is located at head node. The shared policy reported in Figure 2.9-(b) specifies which data elements (handle, jobid, node-id, subjob-id, subjob-node-id, authentication-token, access-control-decision, access-control-policy, process-id, and timestamp) have to be collected by execution of the agent's action, COLLECT-JOB-DATA, when the job state becomes *Pending*. The policy also specifies that agents have to send (agent's action, SEND) the required data (handle, jobid, node-id, and timestamp) to sub-job's destination when state changes to *Active*. The elements to send according to a shared policy are crucial in order to generate the cover-record. The handle is the temporary identifier generated at the IdP or entry point and unique for each job. Since the handle keeps a direct connection to a real user's identity, it is valuable for accountability. When a job travels across multiple domains, the job changes its name. Thus, for the shared-policies defined according to job-flow based approach, the local job identifier is considered an important element. Additionally source information (node-id) from which sub-jobs are sent is of interest for constructing the job-relation graph on cover-record. Finally timestamp is also important element for both shared and local policies to specify when the action is performed. In addition to the elements shown in the example, the policies may also include the retention-time specifying for how long the collected data should be kept.

```
<?xml version=”1.0”    …?>
<AccA_Policy>
<HeadNode>
    <COLLECT-RESOURCE-DATA>
          <data>jobid</data>
          <data>ctime</data>
          <data>qtime</data>
          <data>etime</data>
          <data>cput</data>
          <data>mem</data>
          <data>vmem</data>
          <data>walltime</data>
          <data>cpu</data>
          <ts>timestamp</ts>
    </COLLECT-RESOURCE-DATA>
</HeadNode>
</AccA_Policy>
```

```
<?xml version=”1.0”    …?>
<AccA_Policy>
    <Pending>
        <COLLECT-JOB-DATA>
              <data>handle</data>
              <data>jobid</data>
              <data>node-id</data>
              <data>subjob-id</data>
              <data>subjob-node-id</data>
              <data>authentication-token</data>
              <data>access-control-decision</data>
              <data>access-control-policy</data>
              <data>process-id</data>
              <ts>timestamp</ts>
        </COLLECT-JOB-DATA>
     </Pending>
     <Active>
        <SEND>
              <data>handle</data>
              <data>jobid</data>
              <data>node-id</data>
              <ts>timestamp</ts>
        </SEND>
        <RECEIVE>
              <data>subjob-id</data>
              <data>subjob-node-id</data>
        </RECEIVE>
     </Active>
</AccA_Policy>
```

Figure 2.9 Abstract Representation of (a) Local Policy, and (b) Shared Policy

The shared policy consists of *essential accountability* and *specified accountability*. The essential accountability is the minimum level of accountability required to complete a cover-record. The data elements of essential accountability are *handle, jobid, node-id, timestamp, subjob-id,* and *subjob-node-id* in the example of Figure 2.9. The specified accountability is the accountability level defined to collect specified data in the shared policy. The data of specified accountability are *handle, jobid,*

node-id, subjob-id, subjob-node-id, authentication-token, access-control-decision, access-control-policy, and *process-id* from the Figure 2.9.

We abstract from the underlying policy encoding by using the following simple formalism for the two policy types.

Definition 2.5 (Accountability policies). An accountability policy is an expression of one of the following form:

- A shared policy *shared_policy* is an action expression $AE = \{A_1,...,A_m\}$, specified according to Definition 2.4, such that $\forall i \in [1, m],\ \forall j, k \in [1, n], j \neq k, A_i.Site_j = A_i.Site_k$
- A local policy *local_policy* (among n organizations) is an action *expression* $AE = \{A_1,...,A_m\}$, specified according to Definition 2.4, such that $\exists i \in [1, m],\ \forall j, k \in [1, n], j \neq k, A_i.Site_j \neq A_i.Site_k$

In other terms, accountability policies are action expressions specified for the same agent, as specified in the definition by the condition on the parameter *x* of actions in AE. Local policies have the additional constraint of being expressed only in terms of actions expressing data collection. By contrast, shared policies may include any combination of actions. If collection actions are included, the intended meaning is that the data is shareable with other agents upon request.

Shared and local policies are specified according to the grammar in Figure 2.10 We use the Backus-Naur notation to describe the syntax of the accountability policy language. Our grammar mainly consists of *action_specification*, *Acc_data,* which is *job_flow_based* or *grid_node_based*, and terminal variables such as state, names of data supported in the languages.

We give an example of shared and local policies in what follows.

Example 3. A job is submitted to Purdue University SP and then assigned for execution to the RPs, A-state University, and B-state University. Purdue agrees to send job-relation data (handle, job-id, subjob-id, RP-id, timestamp) to A-state and B-state when the processed job enters into active state. Additionally, A-state locally collects resource data (memory consumption, cpu time, network bandwidth, disk bandwidth) every day during the week.

The policies for such scenario are as follows:

```
<policy_set> := <policies>
<policies> := <policy> <policies> | <policy>
<policy> := <action_specification> | <representatives>
<representatives>:=<Acc_data> <symbol> <representatives> | <Acc_data>
<Acc_data> := <job_flow_based> | <job_assigns> | <strings> |
<boolean> | <pair> | <grid_node_based>
<pair> := (<job_type><symbol><state>)
<resources> := <resource> <period> | <constraints>
<state> := submitted | created | started | completed | pending | aborted |
queued | suspended | active | done
<job_type> := computational | transfer
<job_flow_based> := handle | job_id | process_id | executable | SP_id |
IdP_id | file_names | platform | timestamp
<grid_node_based> := memory consumption | CPU time |
network bandwidth | disk bandwidth | IP_destination | port
<constraints> := all_process | life_time | all_day | weekdays | weekend
<job_assigns> := SP_id ← job_id
<strings> := authorization_policy | usage_policy
<boolean> := authorization_decision
<symbol> := (AND) | (OR)
<period> := : NUM | null
```

Figure 2.10 Accountability Grammar in BNF

[Purdue]

shared_policy$_{\textbf{\textit{Purdue}}}$:=

send_job_data (agent@Purdue, agents_in_job_relation$_{\text{Purdue}}$, active, dataSet$_{\text{active}}$, job-id)

collect_job_data (agent@Purdue, active, dataSet$_{\text{active}}$, DB$_{\text{Purdue}}$)

agents_in_job_relation$_{\textbf{\textit{Purdue}}}$:= agent@A-state (AND) agent@B-state

dataSet$_{\textbf{\textit{active}}}$:= handle (AND) job-id (AND) subjob-id (AND) RP-id (AND) timestamp

[A-state]

local_policy$_{\textbf{\textit{A-state}}}$:=

collect_resource_data (agent@A-state, $dataSet_{local}$, $time_constraints_{A\text{-}state}$, $DB_{A\text{-}state}$)

$dataSet_{local}$:= memory consumption (AND) cpu time (AND) network bandwidth (AND) disk bandwidth

$time_constraints_{A\text{-}state}$:= weekdays (AND) all.days

Example 4. (Continued from example 3) When the resource misusage (memory and CPU) is found at A-state and reported to Purdue, Purdue requests accountability information (handle, job-id, subjob-id, RP-id, timestamp, memory comsumption, CPU time) both to A-state and B-state based upon the agreed contract.

The policies for such scenario are as follows:

[Purdue]

$shared_policy_{Purdue}$:=

request_job_data (agent@Purdue, $agents_in_job_relation_{Purdue}$, $dataSet_{fail}$, job-id)

$dataSet_{fail}$:=

handle (AND) job-id (AND) subjob-id (AND) RP-id (AND) timestamp (AND) memory consumption (AND) cpu time

combine_job_data (agent@Purdue, $agents_in_job_relation_{Purdue}$, $dataSet_{combined}$, job-id)

$dataSet_{combined}$:= $dataSet_{Purdue}$ (AND) $dataSet_{A\text{-}state}$ (AND) $dataSet_{B\text{-}state}$, $dataSet_{Purdue}$:= ***$dataSet_{A\text{-}state}$*** := $dataSet_{B\text{-}state}$

[A-state]

$shared_policy_{A\text{-}state}$:=

forward_job_data (agent@A-state, agent@Purdue, $dataSet_{fail}$, job-id)

combine_job_data (agent@A-state, ∅, $dataSet_{A\text{-}state}$, job-id)

3. PROFILE-BASED SELECTION OF ACCOUNTABILITY POLICIES

When a job is submitted to a node, the accountability agent starts collecting job-related data based on the shared accountability policy. However, although the shared policy specifies the data to collect, some nodes may not have the capability to comply with this policy because of their own limitations, such as insufficient log information, different software versions, and different applications, etc. The different nodes have also different limitations in what they can collect depending on their role in the grid. For example, if the shared policy enforces to collect an element that is only available at a gatekeeper node, the compute node cannot comply with the policy.

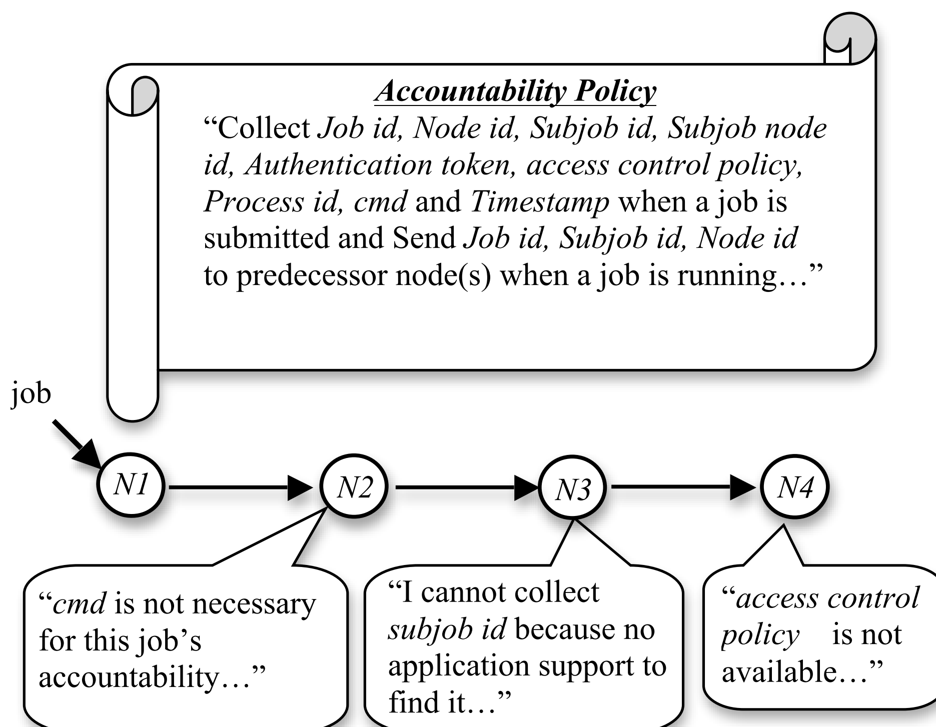


Figure 3.1 Examples of Policy Conflict

An example in Figure 3.1 shows such conflict. When a job is submitted to a node *N1* and then executed at *N4* via *N2* and *N3*, the accountability policy in Figure 3.1 is enforced at each node. Access control data such as the pertinent access control policy or the outcome of its enforcement is also relevant information for accountability. This data is generally collected at the first node to which the job is submitted, to grant or deny the permission to use the grid resources. However if the policy that requires collecting access control data is enforced at node *N4*, which does not involve access control, a conflict arises. When the job is transported to *N3* via *N2* from *N1*, the policy requires collecting sub-job id. However *N3* cannot collect such data because there is no application to support for collecting this data. As a result, *N3* violates the policy and causes a critical deterioration in accountability. When the job is fairly safe, but the node is logging too much detailed monitored data, complex policy should be prevented. For example, in Figure 3.1 if the job is transited to *N2* from *N1*, the policy requires collecting *cmd,* that is, a path and filename of the job to be executed as one of elements to collect in the node *N2*. However, such detailed information is redundant for accountability in *N2* because the actual execution will be performed in *N4*.

What if the accountability policy to be shared is very simple such as *"Collect Job id, and Subjob id and Send them to Subjob's destination"?* This simple policy could lead to insufficient accountability data, therefore resulting ineffective. In summary, as thesis examples show, guaranteeing full accountability while addressing conflicting issues is not a simple problem.

If a node cannot fully support the shared policy, it is still however desirable to collect only mandatory data within its capabilities, satisfying the purpose of sharing policy, rather than aborting the job. To this extent, each agent performs a selection

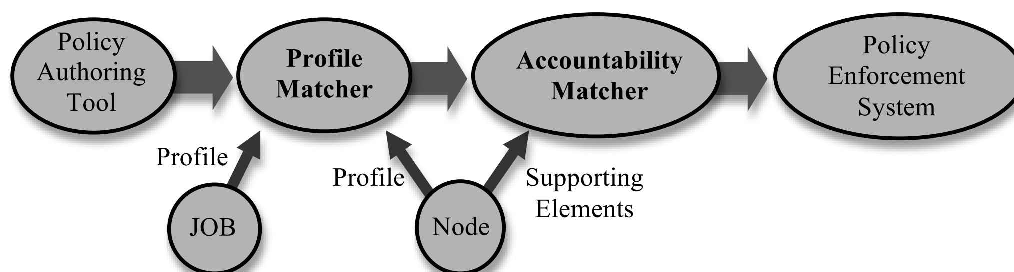


Figure 3.2 The Lifecycle of the Accountability Policies

process for an appropriate set of collectible data. This selection process plays an important role in the design of an accountability policy that will be enforced at each node. The accountability policy sets the *level of accountability* that indicates how much the job and node is accountable by the policy. The higher the level of accountability is, the more accountable it is considered to be. Such policy is influenced by the *job's level of potential risk* as well as the *significance* of the node with respect to the system.

The accountability agent that performs policy selection process is suited with two logical components as shown in Figure 3.2. One is *profile matcher* and the other is *accountability matcher*. In this Chapter, we focus on the policy selection process, that is, on the tasks performed by these two matchers. These two steps are the most challenging and interesting, while the other steps focus on the enforcement of the policies and are common in other policy-based systems.

3.1. Profile Matcher

In grids, jobs are submitted with a description expressed in a job description language [75]. Different types of schedulers provide different job description languages; however, despite such heterogeneity, the description contents are very much the same across the various types of scheduler. The job profile, specified by one of such job description languages, contains information about how many processors and nodes are requested for the job execution, how much running time or memory is required, where the job is coming from, etc. The accountability agent transports this job profile from the entry node to each node where the job or its sub-jobs are assigned or executed. Each node is also characterized by a profile containing information about its hardware, software, and network. The node's profile is specified before jobs are submitted and it is not subject to change. Thus, the agent uses the same node's profile for all jobs. Examples of profiles for job and node are shown in Figure 3.3.

The profile matcher, a component of the accountability agent, maintains two metrics indicating how risky a job is and how important a node is. We refer to such metrics as to *risk factor* and *significance factor*. The job that potentially consumes computing resources by requesting a high number of CPUs or a huge amount of memory

```
Profile for Job A =
{
Type := "Job";
SentFrom := "country A";
NumberOfRequestedCPU := 660;
NumberOfRequestedNode := 300;
RequestedMemory := 24 GB;
InputFile := /home/wlee/input.txt;
WallTime := 720:0:0;
Project := "TG-AIG009382";
}

Profile for Job B =
{
Type := "Job";
SentFrom := "country Z";
NumberOfRequestedCPU := 80;
NumberOfRequestedNode := 40;
RequestedMemory := 128 KB;
WallTime := 90:0:0;
Project := "TG-BWG009386";
}

Profile for Node X =
{
Type := "Machine";
Role := "Gatekeeper";
Name := "gk.rcac.purdue.edu";
Disk := 160;                // giga bytes
Memory := 4000; // mega bytes
LoadAverage := 0.098341;
Arch := "Intel Core 2 Duo";
ProcessorSpeed := 2.16 GHz;
Premium := True;
}

Profile for Node Y =
{
Type := "Machine";
Role := "Compute Node";
Name := "hn.rcac.purdue.edu";
Disk := 140;                // giga bytes
Memory := 1000;    // mega bytes
LoadAverage := 0.022869;
Arch := "Intel";
ProcessorSpeed := 1.28 GHz;
Premium := False;
}
```

Figure 3.3 Example of Profiles For a Job

and is submitted to a critical node, may be malicious and needs to be monitored more closer than other jobs. The introduced risk and significance factors help classify how much accountability data should be collected for a given job in a node. The risk factor is a pair of an element from the attribute set of the job profile and its value. The value specifies how much the element in that attribute of the job is risky. The values are positive real numbers and are same through all nodes for consistent comparison. We initially assume that if the job requests many resources and is submitted from potentially dangerous sites as specified by the administrators, the job appears to be riskier than others. The significance factor considers how important the node is, compared to other nodes. If a node has a special and unique role such as authenticating users and

```
Risk factor for a Job =
{
Type := "Job";
SentFrom := {
        {"country A", "country B", "country C"} = 2,
        {"country D", "country E"} = 1.5,
        Others = 1;}
NumberOfRequestedCPU := {
        {701 ~ 1000} = 2,
        {401 ~ 700} = 1.5,
        {1 ~ 400} = 1;}
NumberOfRequestedNode := {
        {301 ~ 500} = 2,
        {101 ~ 300} = 1.5,
        {1 ~ 100} = 1;}
RequestedMemory := {
        {24GB ~ 32GB} = 2,
        {4GB ~ 24GB} = 1.5,
        {~ 4GB} = 1;}
IsInputFileRequired := {
         "Yes" = 1,
         "No" = 0;}
WallTime := {501 hr ~ 720 hr} = 2,
            {171 hr ~ 500 hr} = 1.5,
            {1 hr ~ 170 hr} = 1;}
}
```

```
Significance factor for a Node =
{
Type := "Machine";
Role := {
            "Gatekeeper" = 2,
            "Service Provider" = 1.8,
            "Head Node" = 1.5,
            "Compute Node" = 1;}
LoadAverage := {
             {0.08 ~ 0.1} = 2,
             {0.03 ~ 0.0799} = 1.5,
             {~ 0.0299} = 1;}
Quality := {"Premium" = 1,
          Others = 0;}
}
```

Figure 3.4 Example of Risk Factor and Significance Factor

authorizing user's requests or scheduling jobs to CNs, such node should be considered more significantly than the ones with a less critical role such as providing only computing cycles or memory. Even for the same role, if a node deals with more jobs than others, the node should be considered more significantly and thus be more accountable than the others. The value of the significance factor can also vary based on the agreement of administrators. The higher the value is, the more accountable the node should be. The range of the factor can span to any size and the classification is determined based on the agreement by administrators. A diverse range gives more fine-grained accountability since it can result in different levels of accountability. The example about risk factor and

significance factor in Figure 3.4 shows two to four classes of the profile for each attribute having values between 0 and 2.

The requested level of accountability for a job, that is, how much detailed data has to be collected for a job, is set based on the job itself and the node profiles. In order to determine such level, the attributes in the job and node profiles are converted into risk and significance factors. The overall level is calculated by multiplying the two metrics. If a job that according to its profile appears to be at high risk is submitted both to a critical node and to a non-critical node, the resulting risk value for this job should take into account the impact on these two nodes.

For example, if a job with a risk factor of *2* is submitted to two different nodes *N1*, *N2*, whose significance factor is *1* and *2* respectively, then the risk that one incurs when submitting the job to *N2* is two times higher than the risk of submitting the job to *N1*. Likewise, if two different jobs with risk factor equal to *1* and *2*, respectively, were submitted to the same node, the job whose risk factor is *2* would have twice as large combined risk as that of *1*. From these observations, we can assume that the significance factor and risk factor are two independent factors that can thus be linearly combined to obtain *combined risk*. We define this combined risk as the *requested accountability*. The requested accountability is proportional to the combined risk, which means the higher the combined risk is, the higher the requested accountability has to be. Therefore the accountability requested for a job submitted to a node can be described as follows.

$$\textit{Requested accountability} = Req_{Acc} = c\sum_{i,j} X_i Y_j \tag{3.1}$$

$$\textit{Normalized requested accountability} = Req_{Acc} / \sum_{i,j} P_i Q_j \tag{3.2}$$

where c[5] is a coefficient, $1 \le i \le n$, $1 \le j \le m$, $\boldsymbol{X}$ is the set of metric elements in the job's profile, $\boldsymbol{Y}$ is the set of metric elements in the node's profile, $\boldsymbol{P}$ is the set of highest metric elements in the job's profile, $\boldsymbol{Q}$ is the set of highest metric elements in the node's profile, $\boldsymbol{n}$ is the number of elements in $\boldsymbol{X}$ and $\boldsymbol{P}$, $\boldsymbol{m}$ is the number of elements in $\boldsymbol{Y}$ and $\boldsymbol{Q}$

[5] We consider coefficient as 1 for simplicity from now

The highest value in the range of the requested accountability levels is obtained by the multiplication of the highest risk factors by the highest significant factors, and the lowest value is from low risk factors by least significant factors. Examples of requested accountability for jobs A and B at Node X and Y of Figure 3.3 are given below.

Example 1. Consider Job A at Node X. Requested accountability=[{2,2,2,2,1,2}*{2,2,1}]/[{2,2,2,2,1,2}*{2,2,1}]=((2+2+2+2+1+2)*(2+2+1))/((2+2+2+2+1+2)*(2+2+1))=1. Job A that has many highly risky factors is submitted to a node classified as most significant node, i.e. Node X. By equation 3.1 and 3.2, the calculated risk value requires a highest accountability level; thus when Job A is submitted to Node X, the accountability agent located at Node X needs the fully requested accountability, which is the highest level of accountability for Job A.

Example 2. Consider Job B at Node X. Requested accountability=[{1,1,1,1,0,1}*{2,2,1}]/[{2,2,2,2,1,2}*{2,2,1}]=((1+1+1+1+0+1)*(2+2+1))/((2+2+2+2+1+2)*(2+2+1))=0.455. Even though Node X is significant, because Job B does not have highly risky factors, only 0.455 worth of accountability level is requested.

Example 3. Consider Job B at Node Y. Requested accountability=[{1,1,1,1,0,1}*{1,1,0}]/[{2,2,2,2,1,2}*{2,2,1}]=((1+1+1+1+0+1)*(1+1+0))/((2+2+2+2+1+2)*(2+2+1))=0.182. Since Job B in Example 2 is submitted to a less significant node (i.e. Node Y) than Node X, the risk is lower than the one obtained in

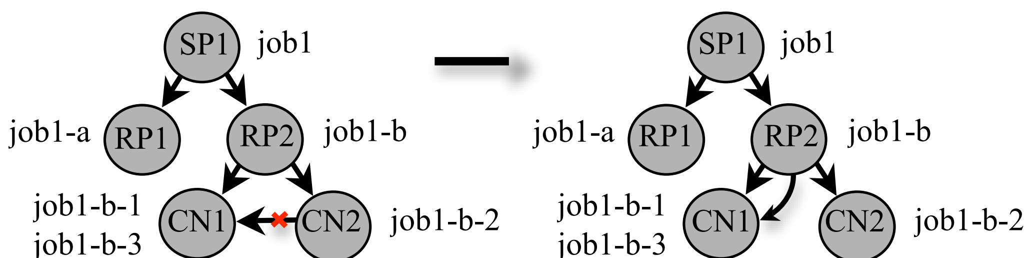


Figure 3.5 (a) An Incomplete Job-graph Due to Insufficient Accountability
(b) The Reconstructed Job-graph

Example 2. The agent at Node Y needs the minimum requested accountability, which is the lowest level of accountability for Job B.

3.2. Accountability Matcher

Since each node has different constraints for collecting elements specified in a policy, the level of support for accountability is different from node to node. Here, we define the level of accountability that can be supported by each node as the *supported accountability*. While the requested accountability is generated for each job, the supported accountability is statically defined for each node. Definitions of accountability levels and related terminologies are listed in Table 3.1.

By integrating and coordinating the requested and supported accountability, the accountability matcher selects the best policy based on the shared accountability. The shared policy specifies the elements required for the agent to complete the cover-record as introduced in Chapter 2. If these elements are not available at the node, then the agent has *insufficient accountability.* Therefore, the cover-record cannot be created, thus resulting in an incomplete job-graph (Figure 3.5-a). In such case, the node that gives insufficient accountability is dangling in the job-graph and does not connect any successor node. To reconstruct the job-graph, the agent in the dangling node sends all information about the cover-record received from the direct predecessor node to the direct successor nodes (See Figure 3.5-b for a reconstructed job-graph). For example, if *node-id* is not available in *CN2* and cannot be sent to *CN1*, the agent in *CN1* cannot send *job1-b-3* to *CN2* due to missing address of the node represented as *node-id*, thus resulting in a lost connection. In this case, the agent in *CN2* forwards {*handle*, '*job1-b*', '*RP2*', and *timestamp*} received from *RP2* to the agent in *CN1*, which is the direct successor of *CN2* to reconstruct the job-graph. Though the connection between *CN1* and *CN2* for the job '*job1-b-2*' is lost, the accountability for the jobs '*job1-b-2*' and '*job1-b-3*' is still guaranteed because *CN2* and *CN1* are still connected to *RP2* in the job-graph for the '*job1-b-2*' and '*job1-b-3'* respectively.

Table 3.1 Definitions of Terminologies Used by Matchers

Terminology	Definition
Risk factor	The degree that shows how risky a job is
Significance factor	The degree that shows how important a node is
Price	The degree of importance in terms of accountability for accountability data
Essential accountability	The minimum accountability level of shared policy required to complete a cover-record
Specified accountability	The accountability level defined to collect specified data in the shared policy
Supported accountability	The level of accountability that can be supported by a node. The sum of prices put on elements
Requested accountability	Linearly combined risk of risk factor about a job with significance factor about a node

In a significant node, applications such as grid middleware and job-schedulers, that make the node significant, provide high possibility for collecting job-related information. For example, in a gatekeeper where the Globus Toolkit (GT4) [18] is installed, the agent can collect job status, node-id, and subjob's id information directly from Globus. In a head node where the job scheduling is performed, the agent can obtain some useful accountability information, such as sub-job ids assigned at each CN, and *cmd* from the scheduler's log file. From this observation, it is thus clear that the possibility of having sufficient accountability in a significant node is very high, while it is low in a less significant node, such as a computational node. Thus the CN2 in Figure 3.5-b has high probability that it is less significant node expecting a low accountability.

The first task of the accountability matcher is to compare the requested accountability transmitted from the profile matcher with the supported accountability. The supported accountability is computed by summing up numeric values put on each collectible data element in a node. We define such numeric value as *price*, which means a degree of importance in terms of accountability. The higher the value of the price is, the more important the element is for accountability. There are several criteria to decide the degree of importance of accountability for each element. First if the element is essential to construct a job-relation, it has a high price because it provides provenance data concerning the executed jobs, which is crucial information in accountability. Second, the

data obtained from the job-flow based strategy has higher price than elements collected from grid-node based strategy. Tracing back a job and its action across various nodes is one of important tasks of accountability system. Data obtained from multiple nodes in job-flow based strategy fulfill this task better than data obtained in a fixed node under the grid-node based strategy. Third data related to security such as authentication tokens, and access control decision or policy has high price. Because for example, access control determines which jobs are executed on the grid, access control decision is crucial to determining if and why wrong decisions were made. Through the three criteria in the specified order the unit price that is identical for all nodes, is put on the elements upon agreement of administrators. The detailed process about determining the price for each element is out of scope of this thesis. The goal of using price is to show a level of accountability a node can support for requested accountability in a number.

Each node owns a list of elements that can be collected with a summed price. When summing up the prices for supported accountability, a constraint is enforced. If any of elements required to construct a cover-record for the original shared policy is not supported, other elements besides these required elements cannot be summed up. This constrains is to guarantee the essential accountability of the shared policy.

Table 3.2 shows examples of elements with different cases and prices, (A) through (G), that a node can support. For example, if a node supports, only *Handle, Job-Id, {Subjob-Id, Subjob-node-Id},* and *TS,* which are the elements to satisfy the essential accountability, the supported accountability of the node becomes 0.582 summed of all priced values – case (B). For case (G), the accountability is fully supported. The range of the supported accountability is the same as the requested accountability (i.e., *Minimum supported accountability = Minimum requested accountability = 0.182 < {Supported accountability, Requested accountability} < Fully supported accountability = Fully requested accountability = 1)* so that the accountability matcher can compare them. For each comparison between the supported accountability and requested accountability, the *shared accountability* represented as bidirectional arrows (1) through (5) in Figure 3.7 is compared again. Each arrow spans from the level of essential accountability (i.e., the left-

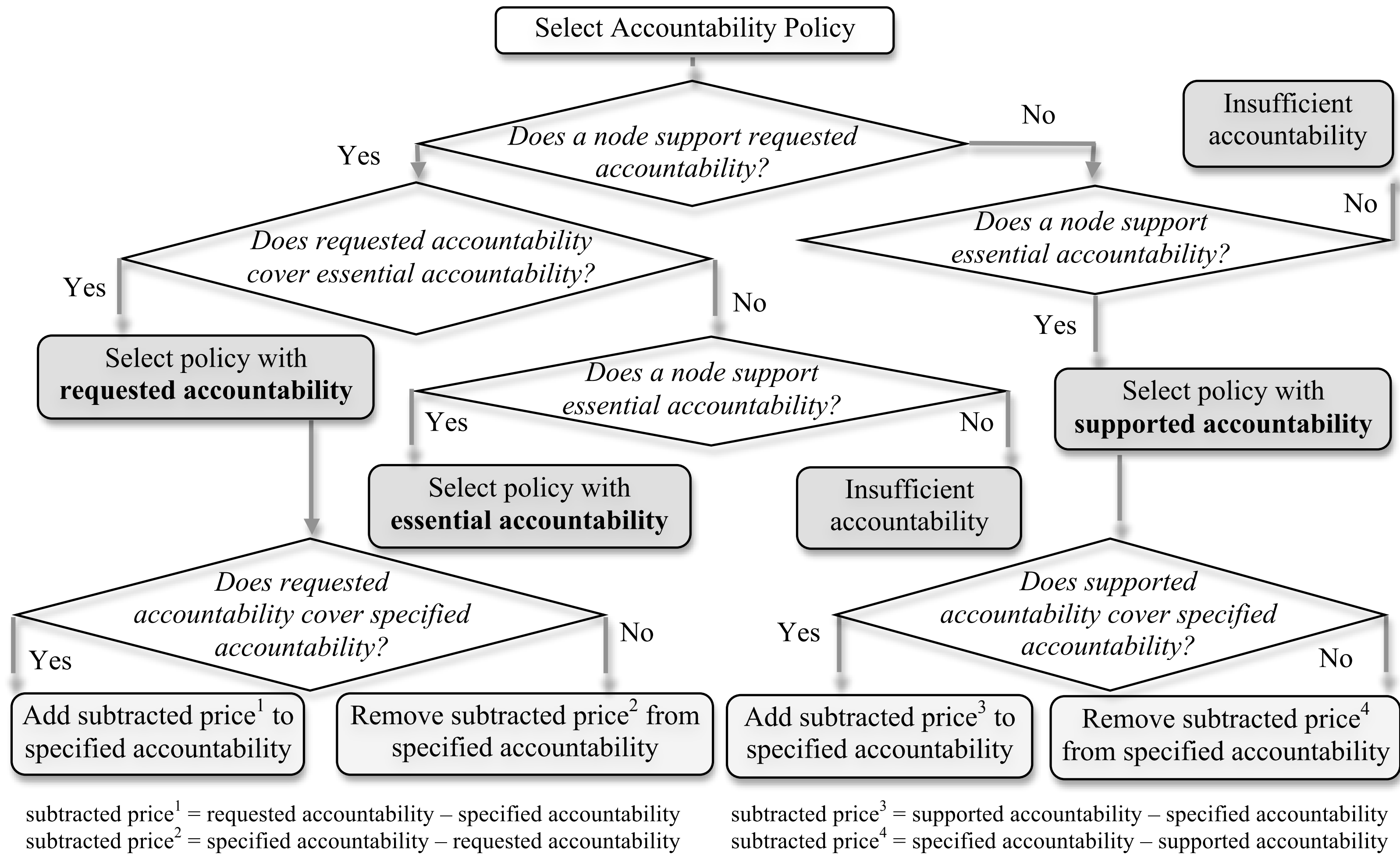


Figure 3.6 Flow Chart Diagram for Selecting Shared Accountability Policy

most of the arrow) to the level of specified accountability (i.e., the right-most of the arrow). Depending on the actual data being collected at the node, the size of the arrow varies. For example, it can be smaller than requested accountability or supported accountability like examples A-(1) and B-(1) in Figure 3.7 respectively. Or, the range of the shared accountability is in between requested accountability and supported accountability like examples A-(3) and B-(3).

In case that a requested accountability is fully supported (case A in Figure 3.7) in the node, the requested accountability is selected as long as it covers the essential accountability. This is the case A-(1) and A-(2) where the essential accountability is satisfied by selecting the requested accountability. The difference between the requested accountability (A-x) and the specified accountability of A-(1) is incorporated with the price that should be added to the shared policy by increasing the specified accountability to much of subtracted price. In case of A-(2), the difference between the specified accountability and requested accountability (A-x) is also incorporated with the price that should be removed from the specified accountability. This is because we only need the requested accountability. The overhead that occurred by collecting data represented as the price of such difference can be prevented.

If the essential accountability is not guaranteed due to the selection of policy with the requested accountability as the case A-(3) in Figure 3.7, instead of the requested accountability, the essential accountability is selected by the accountability matcher. In this case, since selecting the policy with the essential accountability satisfies the requested accountability, nothing is added or removed from the selected policy. For the case, A-(4), even though the specified accountability is not supported, if the policy only with the essential accountability is selected, the policy satisfies the requested accountability, thus nothing is added or removed from the selected policy. When the shared accountability has a range of accountability level such as A-(5) – i.e., essential and specified accountability are not supported, the node is dangling in the job-graph and the modified policy from the shared policy will be applied.

In the case of a node that cannot support the requested accountability (case B), if the shared policy has the level of accountability with the range B-(1), selecting the policy

Table 3.2 Supported Accountability (Pr: price, ACL: Access Control, PID: Process ID, TS: Timestamp, Attr: Attributes, cmd: file name and path of executable)

	Handle	Node-Id	Job-Id	ACL-Decision	ACL-Policy	Subjob-Id	Subjob-node-Id	PID	Job State	TS	Auth Token	Attr	cmd	*Supported Accountability*
Pr	0.091	0.1	0.091	0.075	0.075	0.1	0.1	0.075	0.025	0.1	0.093	0.025	0.05	
(A)	√		√											*0.182*
(B)	√	√	√			√	√			√				*0.582*
(C)	√	√	√			√	√		√	√	√			*0.7*
(D)	√	√	√	√	√	√	√	√		√				*0.807*
(E)	√	√	√	√	√	√	√	√		√		√	√	*0.882*
(F)	√	√	√	√	√	√	√	√	√	√		√	√	*0.907*
(G)	√	√	√	√	√	√	√	√	√	√	√	√	√	*1*

with supported accountability satisfies the specified accountability as well as the essential accountability. This selection makes the new shared policy close to the requested accountability. If the shared accountability is in the range (case B-(2)), the selection of supported accountability satisfies the essential accountability and specified accountability to the level that the node can support (i.e., to B-x from the essential accountability). If the essential accountability is not supported as in case B-(3),(4),(5), these cases become insufficient accountability and the corresponding resolution technique is applied as described above. This node will have the policy modified from the original shared policy. The modified policy will have the same elements as the original one for the essential part and will change *SEND* to *FORWARD* in Figure 2.9-b, without the need to collect data since the data has already been received from the predecessor node. The following examples show the selection process in terms of price.

Example 4. A profile matcher requests accountability at Node Z for Job C to be 0.85. Assume that Table 3.2-F is the supported accountability at Node Z (i.e. supported accountability = 0.907).

For case A-(1) in Figure 3.7: If the essential accountability is Table 3.2-B (i.e. 0.582) and the specified accountability is Table 3.2-C (i.e. 0.7), which is less than the requested accountability (i.e. 0.85), the policy with 0.85 is selected and the elements of which whole price is the difference between A-x (0.85) and the specified accountability (0.7) (i.e. 0.15 = 0.85 – 0.7) should be added to the shared policy. In this case, candidates that can be added are selected from the supported elements, i.e., from the list Table 3.2-F.

Thus {“ACL-Decision”, “ACL-Policy}, or {“PID”, “Attr”, “cmd”} priced altogether 0.15, could be added to the shared policy.

For case A-(2) in Figure 3.7: If the essential accountability is Table 3.2-C (i.e. 0.7) and the specified accountability is Table 3.2-E (i.e. 0.882) that is greater than requested accountability (i.e. 0.85) and less than supported accountability (i.e. 0.907), then the policy with 0.85 is selected and the elements of which whole price is the difference between the specified accountability, A-(2) (i.e., 0.882) and A-x (0.85) (i.e. 0.032 = 0.882 – 0.85) should be removed from the shared policy except elements in Table 3.2-C. In this case, because there is not an element that exactly matches to 0.032 and the specified accountability is lower than the supported accountability, the highest priced element out of ones smaller than 0.032 is chosen to be removed such as {“Attr”} priced 0.025.

For case A-(3)(4) in Figure 3.7: If the essential accountability (i.e., Table 3.2-E) and specified accountability of the shared policy are greater than the requested accountability (i.e., 0.85), the policy with the essential accountability (i.e., 0.882) is selected.

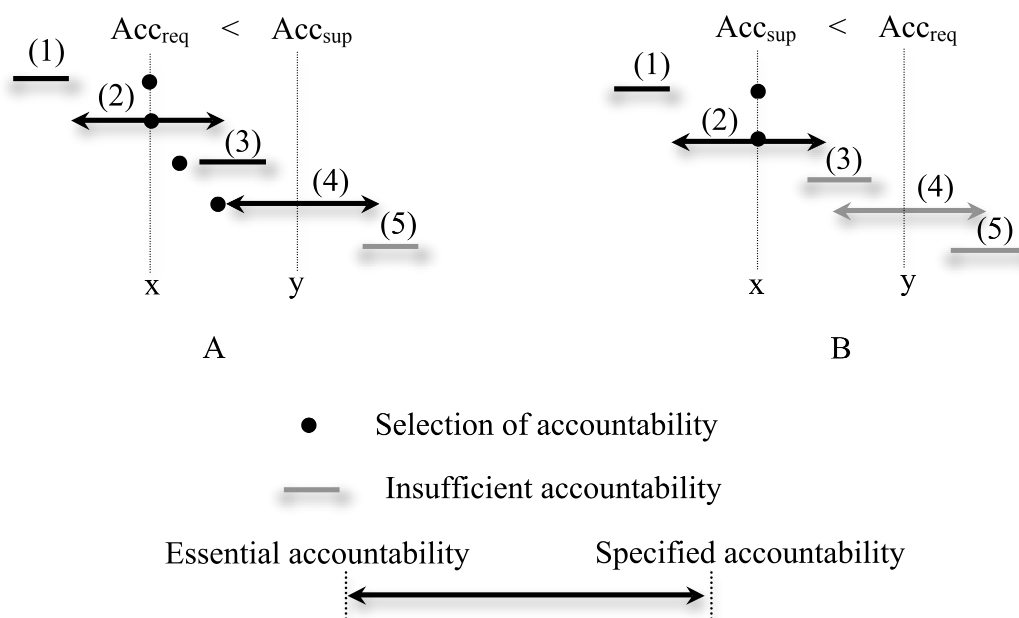


Figure 3.7 Cases of Comparisons with Shared Accountability

4. VULNERABILITIES IN GRID COMPUTING SYSTEMS

Because of the scalable and dynamic nature of the grid, and the lack of grid-specific security protection mechanisms, grid systems suffer from several vulnerabilities. Vulnerabilities can be found at each grid layer and can be exploited by an intruder to bypass the system's authentication or authorization, or by malicious user's code submitted as part of a grid job. Attackers can also take advantage of grid resources to launch distributed denial of service (DDoS) attacks, or to crash one of the grid components, resulting in the grid outage. If, for example, a head node of a cluster where the actual grid job scheduling is performed is attacked and cannot execute its normal functions, all computing resources connected to that head node will not be available to legitimate users. If the web-services running at gatekeeper of a RP are denied due to a DDoS attack, the legitimate users' requests to that gatekeeper cannot be transferred to other gatekeepers or to the scheduler. Attackers can also target servers located outside the grids such as mission-critical government websites or popular commercial hosts.
In this chapter an overview of the most common grid vulnerabilities are presented. The following sections describe possible or known vulnerabilities, classified according to their locations in the grid layers from the low to the high layers.

4.1. Vulnerabilities of the Connectivity Layer

GSI [11] provides a set of fundamental security services that are specifically designed to support grids. GSI relies on certificates to handle authentication [11]. In a GSI certificate, there are four important elements: a subject name; a public key that belongs to the subject; the identity of a certificate authority (or CA) that has signed the certificate; and digital signature of the CA [11]. If the CA that signed the certificate is not

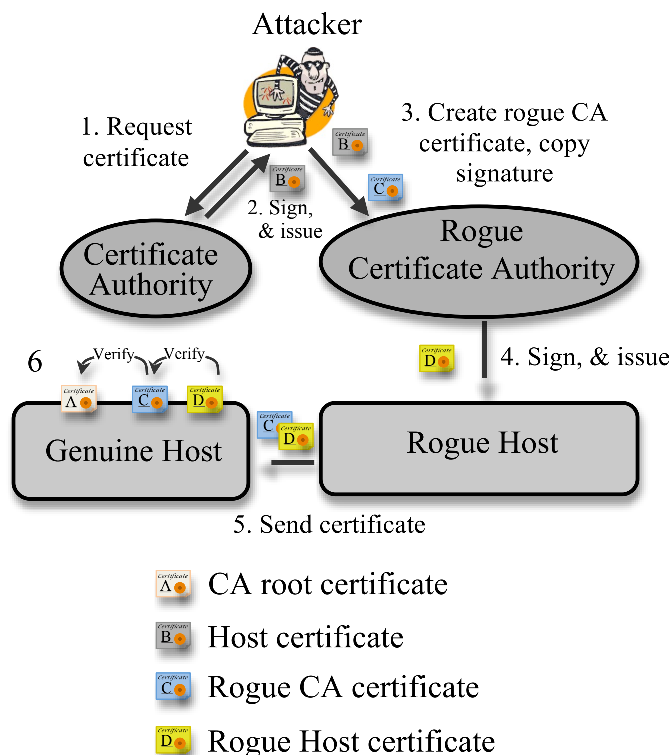


Figure 4.1 Attack Scenario by MD5 Collision

genuine but has a real signature of the genuine CA, a malicious user can manipulate a GSI certificate signed by that CA. Stevens et al. [33] report that it is possible to hash two different messages to the same MD5 hash value by MD5 cryptographic hash function. Using the collision in the MD5 hash function in digital signatures that can lead to an attack against the GSI, the rogue CA creates a rogue CA certificate. This certificate is trusted and accepted by all common hosts providing grid resources since it bears a valid signature singed by the genuine CA.

Figure 4.1 shows an attack scenario that exploits the MD5 weakness discussed in [33]. The main steps are as follows.

1. An attacker who wants to exploit grid resource searches a CA that uses the MD5 hash function to generate the signature of the certificate and requests a host certificate if such CA is found.

2. A commercial CA signs the legitimate host certificate and issues a host certificate (the gray one tagged as B).

3. The attacker creates a rogue CA certificate (the blue one tagged as C), and then copies the signature obtained in step 2 to the rogue CA certificate. Therefore certificate C appears as being issued by the CA though the CA has never signed it. This rogue CA certificate is an intermediary CA certificate that can be used to sign other host certificates the attacker wants to issue. Because MD5 hashes both the legitimate and the rogue certificate to the same signature, the digital signature signed by the genuine CA can be copied to the rogue CA certificate resulting in making the rogue CA certificate remain valid.

4. The rogue CA creates a rogue host certificate that bears the legal host's identity but another public key, and signs the created certificate to issue to a rogue host.

5. If two hosts have certificates, and they trust the CAs that signed each other's certificates, then the two hosts *mutually authenticate*. Before the mutual authentication is carried out, the rogue host sends the issued rogue host certificate with the rogue intermediary CA certificate to the genuine host to make the genuine host verify the issued rogue host certificate from the rogue intermediary CA certificate.

6. The genuine host verifies the signature of the rogue host's certificate with the rogue CA certificate. The signature of the rogue CA certificate is verified with CA root certificate. The genuine host is therefore lured into trusting the rogue host.

After successfully compromising the host, the attacker can crack the PEM pass-phrase that protects the user certificate in order to access grid resource by using a publicly known tools.

The potential attack exploiting MD5 vulnerability could occur in any Globus [18]-enabled grid system. Globus Toolkit [18], the de-facto standard for grid middleware, is a web-service container for grid services. It provides protocols and services spanning multiple layers of the grid. Globus Toolkit 4.2.1, 4.0.8, and earlier versions use MD5-based signatures in proxy certificates for authentication. In addition, gLite [34], which is another grid middleware developed by collaborative groups of academic and industrial research centers as part of the EGEE [35] Project uses MD5. Some versions of virtual organization membership service (VOMS) [36] included in both gLite 3.1 and gLite 3.2 use MD5 hash function.

The best way to mitigate the risk of this type of attack is to use other types of strong hash function such as SHA2 by updating with the new release of version for Globus Toolkit and gLite.

Another vulnerability related to GSI has been reported in GSI-enabled OpenSSH. GSI-OpenSSH is a modified version of OpenSSH that includes supporting authentication and delegation and is included in Globus Toolkit. The OpenSSH versions prior to version 5.0 contain locally exploitable security vulnerabilities. When a personal computer needs graphical access to a computer on another network, it runs some applications that allow graphical information to pass through firewalls by using a feature called *X11-forwarding*. Unprivileged local users can hijack the X11-forwarded connections by listening on port 6010 when IPv6 is enabled on the server [37]. From the example in [37], assume that a malicious user listens on port 6010 in a certain server by using *netcat* and another user logs in to the same server in order to use *emacs* on the remote system with X11-forwarding. In this case, OpenSSH fails to listen on port 6010 with IPv4 because *netcat* is listening on that port. The OpenSSH however does not try to use other ports since the IPv6 is enabled. Then the OpenSSH sets DISPLAY to ":10" which is set to the malicious user and the *emac* sends cookie to 127.0.0.1:6010. As a result, the malicious user can eavesdrop what the remote user does [38].

4.2. Vulnerabilities of the Resource Layer

In many cases, vulnerabilities are a result of the way the software has been written. Such consideration also applies to the software, which enables secure integration and access to the distributed computing resources owned by different providers. Many vulnerabilities in middleware for grid and parallel computing systems have been reported [39]. GLExec [40] is a standalone executable that maps a grid identity to a Unix/Linux identity. GLExec allows a grid system to execute a user's job so that it is isolated from the grid middleware and from other user's jobs. Vulnerabilities in this software result from a software design error that improperly allows users to specify the name and the location of the log file as reported by Kupsch et al. [41]. Kupsch and colleagues describe that the log file is used by some libraries (i.e., LCAS and LCMAPS) and opened with

root privileges. An attack scenario from [41] is as follows. The attacker specifies the name of the log file as */etc/passwd* in the environment variable and uses some environment variable (for example, LCAS_DB_FILE) whose content can be appended to the end of log record. This small amount of crafted data contains a new line with a valid password and user id as 0 (i.e., root), and group id as 0 (i.e., root) to inject into the log file, which has been changed to the password file. In this scenario, the attacker can gain access to other accounts including the root user.

Some vulnerabilities in the protocol of the Globus Toolkit that monitors and controls computation on the grid resources (i.e., GRAM) have also been reported [12]. When a GRAM job is submitted, the globus-job-manager opens and listens on three temporal ports. Two of these ports are known to be vulnerable. If a remote attacker requests these ports for a GRAM job or its MPICH-G2 applications by sending multiple specially-crafted messages, all the available physical and swap memory can be consumed eventually causing the kernel panic and halting the system as a result [42].

Another vulnerability caused by incomplete sanity check, has been found in Globus Toolkit RFT (reliable transfer service) and MDS (monitoring and discovery system). Multiple local temporary files allow local users to create or overwrite arbitrary files with elevated privileges or to view sensitive information [43]. For example, a generated proxy certificate by default is stored in the */tmp* shared directory. When such sensitive files are generated in a shared directory, the process ensures if the file being written is really created by itself and checks that the file has correct permissions. However it has been reported that some file handling procedure in the Globus Toolkit does not perform the above checks. Attackers can exploit such vulnerability by creating a temporary file with permissions allowing open access. If an attacker has a permission on */tmp* and knows the identifier of the process that creates the temporary file and then links a temporary file to a proxy file, the attacker can access the proxy file and use it for malicious purpose.

4.3. Vulnerabilities of the Collective Layer

The collective layer contains protocol, services, and APIs that captures interactions across collections of resources [10]. Examples of the collective services are resource discovery, scheduling of tasks on the appropriate resources, monitoring and diagnostics services, data replication services, community authorization, certificate revocation, etc. Intruders can also exploit vulnerabilities caused by software design errors in the middleware being used for such collective services. They are for example, grid schedulers such as the Sun Grid Engine (SGE), Condor-G, PBS Pro, or parallel computing software, or credential management software.

Condor-G [14] is a job-submission agent that runs user's grid jobs on the multi-domain resources as if they all belong to one domain. Some software bugs leading to buffer overflow vulnerabilities have been found in Condor-G [41]. An unprivileged local user can gain elevated privileges by exploiting these vulnerabilities. Specifically, there are two potential buffer overflows in the function *Accountant::GetResourceName* in the file *Accountant.C* [41]. The function looks up two attributes (*Name*, and *StartdIpAddr*) whose values are located in two 64-byte buffers. Because users can change the value of these attributes by calling *condor_advertise*, attackers can set these values to overflow the stack.

PBS Pro is another type of software used to schedule grid jobs like Condor-G. By exploiting the vulnerability such that PBS Pro creates temporary files in an unsecure manner, an attacker with local access could perform symbolic-link attacks. An execution demon, *pbs_mom* of PBS Pro uses a world writable directory */var/spool/pbs/spool* for storing jobs' standard output and standard error files. The *pbs_mom* checks whether the file name that it will create with the user's UID and GID exists in the directory. If a file with the same name exists, the file is overwritten. Because the attacker can guess the user's temporary file name, the attacker creates a symbolic link to the file that he created for the guessed temporary file. When the attacker's temporary file is overwritten by the job's standard output and error streams, the attacker can gain access to the user's file with a local access for the link [44]. With this access right, the attacker can delete or corrupt

sensitive files, which may cause a denial of service, by exploiting unsecure temporary files.

The message passing interface (MPI) is an application programming interface (API) for parallel programming used in grid computing systems. The MPI runtime environment for Mandrake Linux is prone to an insecure account creation vulnerability that allows an attacker to create an account '*mpi*' with no corresponding password during installation.

MyProxy also has been reported to be vulnerable, adding one more significant vulnerability of the collective layer. MyProxy is included in Globus Toolkit for managing X.509 public key infrastructure (PKI) credentials. MyProxy allows users to store and manage short-lived X.509 certificate by combining an on-line credential repository with a certificate authority. Different types of vulnerabilities that lead to denial of service attacks are found in the MyProxy.

The following discussion about the MyProxy vulnerabilities is based on the work by [41]. If a client tries to connect to the MyProxy server, the server forks a copy of the server to handle the request. However the forked server can be potentially delayed due to three reasons: lack of time-outs on reads and writes; lack of limits on the amount of data read; and potential deadlocks with child processes. Such vulnerability of the MyProxy server leads to denial of service attacks. After opening a connection, if the client does not send data in the middle of communication with the forked server, the server will wait forever for the data to arrive or until the client closes the connection. The server will clearly waste operating system resources such as processes, memory, and network sockets. The second vulnerability relates to the lack of limitations on the amount of data that the server reads. As a part of a TLS stream, all the data is transmitted as a packet and is encrypted by the client and decrypted at the server after concatenating all packets. The server consumes buffer space to store the decrypted text. However, because the size of the data is not transmitted in advance, the server continuously takes packet streams. If an attacker sends multiple big sized data, for example giga-bytes of data at a time, the server will be out of service. The potential deadlock between the forked MyProxy server and the child process spawned by the server can also lead to denial of service attacks. The

MyProxy server can invoke external programs by calling function *myproxy_popen* in certain configurations. This function returns standard file descriptors such as *stdout, stdin*, and *stderr* connected to the spawned process. The steps for using this function are as follows: (1) writing to the *stdin*; (2) closing *stdin* and waiting for the spawned process to exit; and (3) then reading the data from *stdout* and *stderr*. If the process writes data in the pipe that exceeds the specified limit, the spawned process will be blocked while writing on the read file descriptor but the myProxy server will not read data until the spawned process exits thus resulting in deadlock. When a malicious user who can connect and authenticate to a myproxy-server, crafts a set of parameters in a particular configuration, the availability of the myproxy-server decreases.

The software design errors that cause such reported vulnerabilities in the middleware related to the grid have been timely fixed, and the appropriate development teams have released patches. However, as the possibility that causes other vulnerabilities always exists due to the unknown errors or lack of code validation and verification, the software should be kept up to date in order to defend attacks such as those discussed in this Section.

4.4. Vulnerabilities of the Application Layer

The attackers can exploit vulnerabilities in grid web-services. Grid security incidents related to web-services are reported in [45]. To by-pass site security, attackers use known hacking techniques specific to web-services, such as web-services description language (WSDL) probing, SQL injection attacks, XML attacks, etc.

WSDL probing (or scanning). The web-services advertise their capabilities in WSDL by describing methods and parameters needed to access a specific web-service. A WSDL file is a major source of information for an attacker. The attacker scans the WSDL interface to get sensitive information such as invocation patterns, underlying technologies, and associated vulnerabilities. The WSDL probing is the first step to perform more serious attacks such as parameter tampering, malicious content injection, etc.

The WSDL is often generated automatically in tools such as Java2WSDL. Using such tool, methods in a class or interface are exposed as web-services. Due to automatic generation of WSDL, some critical functions in applications not intended for public use can be converted to web-services unintentionally. Attackers can gain access to private methods by scanning WSDL.

As another WSDL attack, attackers use naming conventions (i.e., get, update, execute, show, etc.) to find the names of methods that are not published in the WSDL but available on the server. For example, suppose that a service that provides climate modeling and simulating service publishes query methods such as *listClimateSimulationCase* in WSDL. When there is an unpublished method but only available on the server such as *executeClimateSimulationCase*, the attacker can discover unpublished application programming interface by guessing in the naming conventions and access to private data and functionality.

SQL injection attacks. Belapurkar and colleagues [46] mention that, according to their real-world experience, web-services typically have higher risk of injection attacks than web applications as services are exposed in human-readable interface formats, making it easier for attackers to inject fraudulent requests. If the server providing the services does

```
<simulationList>
    <user_code> X&apoa; OR 1=1 --</user_code>
    <simulation_type>P</simulation_type>
```

```
String sql = “Select case_name, configuration, creation_time, job_status,
Queue_name, wall_time”
From Simulation
Where user_code = “ ‘+SimulationRequest.getUserCode()+’ ” And
user_status=‘C’
```

```
Select case_name, configuration, creation_time, job_status, Queue_name,
wall_time
From Simulation
Where user_code = ‘X’ OR 1=1 - -’ And user_status=‘C’
```

Figure 4.2 An Example of SQL Injection Attacks in Grid Web-services.
Box A, B, C from Top to Bottom.

not correctly validate the input data, attackers can use a SOAP message to create XML data that inserts a parameter into an SQL query and executes it with the rights of the web-services. For example, suppose that an organization has built a grid based simulation model system and exposed a set of services for simulation case management to its members. Figure 4.2-A shows a web-service request. From this request, a user can see all cases submitted by him. Figure 4.2-B shows the original execution of web-service request. Then the data from the web-service request is replaced and the final SQL represented to the database would be Figure 4.2-C. Because most of the database servers consider "--" as comment, only "*user_code = 'X' OR 1=1*" before "--" is considered to be the condition in the *Where* clause and makes the condition always *TRUE* due to "*1=1*". From this attack, the attacker can see all cases being simulated by other users.

XML attacks. XML has become the de-facto language for interaction among applications. XML includes an element, CDATA defined as unparsed character data. CDATA allows the use of illegal characters in its field since the text data in the field is ignored by the XML parser. Suppose that the XML document is processed to generate an HTML page. If an attacker provides an input such as the example in Figure 4.3-A, the CDATA section delimiters are eliminated during the processing without inspecting their contents. The HTML tags are included in the generated page as shown in Figure 4.3-B bypassing the existing sanitization routines. From this scenario, the application that runs XML with CDATA is vulnerable to cross-site scripting (XSS) attacks [46].

Attackers who want to send possible system commands to the underlying systems use this CDATA element resulting in potential disasters. When querying a XML parser, the CDATA component is removed, and the dangerous characters are generated in the script as shown in Figure 4.3-C and 4.3-D.

XML denial of service (XDoS) attack is another form of XML attack. Attackers carry XDoS attacks to make the services unavailable to legitimate users by flooding the services with huge numbers of requests. XML allows one to use complex nested payload representations. However when attackers intentionally increase the nesting level, such complex payloads lead to a high consumption of resources for parser. Finally a complex recursion of elements crashes the parser. Attackers can also use an alternative strategy to

create DoS attacks. Instead of using complex payloads, they flood the parser by sending a huge message payload. A common method is to convert the creation of an <any> element defined as unbounded to a largely automatic operation. By this technique, attackers can create an unlimited number of elements and crash the parser.

In order to protect from attacks related to malicious input and attacks against XML, XML firewalls can be used. XML firewalls provide functionalities such as checking data authenticity, integrity and validity when inspecting SOAP messages [47].

Grid portals are also possible targets. For example username enumeration attacks [48] have been reported for grid portals deployed by old versions of GridSphere [49], used as a front-end to the TeraGrid [7]. Another weakness of the portal application of GridSphere is the use of form-based authentication by default. This type of authentication conveys the submitted credentials simply as part of the HTML or XHTML <FORM> data; it thus requires encrypted transmission. Although GridSphere can run on HTTPS, it

Figure 4.3 Examples of XML Attack by Using CDATA.
Box A, B, C, and D from Top to Bottom

does not require such a configuration and it is thus exposed to HTTP non-encrypted communications in its default configuration [50]. Attackers can exploit this vulnerability by capturing the HTTP message to impersonate legitimate users. While currently there seems to be no reported problem for this issue, the grid portal equipped with GridSphere should convert the default configuration to run the site on the HTTPS by installing a certificate on the server. The community should no longer accept HTTP to exchange the information.

In terms of authorization, attackers can exploit default configurations of access control for the newly installed web application [50]. Many default configurations include default administrative accounts with either simple passwords easy to crack, or they allow everybody to access. In addition, the administrators mostly control access through user-centric identities or resource-centric capabilities. Although this approach works correctly when the set of users and resources is very simple, when the number of users and resources increases, it is very difficult to manage the access control lists. As a result, a poorly managed access control system can grant low authorization level users access to resources that only high authorization level users can access. To resolve this complexity issue, effective access control mechanisms, such as role-based access control (RBAC) and attribute based access control, should be applied. By assigning users to roles and roles to privileges under RBAC, administrators can effectively give authorizations to users in a fine-grained way. Some researchers have already tried integrating RBAC to existing grid computing systems as reported in [51]. In attribute-based access control, users express their rights by using attributes such as their affiliation, roles in groups, locations, etc. Under attribute-based access control systems such as GridShib [8], although the grid system scales in terms of number of users and resources, the complexity issue is addressed.

5. DETECTION AND PROTECTION AGAINST DISTRIBUTED DENIAL OF SERVICE ATTACKS

By taking advantage of the distributed nature of the grids by processing in parallel and in a short time multiple jobs, the attacker can use resources at critical servers, such as grid schedulers or gatekeepers of resource-providing entities located inside grids, and congest popular commercial and governmental websites located outside the grid, by launching DDoS attacks. Such type of attacks makes grid resources unavailable to legitimate users.

The accountability agents leverage accountability data obtained from the two strategies introduced in Chapter 2 to detect suspicious patterns with the help of existing intrusion detection techniques. The detection takes advantage of certain unique aspects that characterize the behavior of jobs running in grids. In this chapter, we show how a distributed accountable grid computing system can help in detecting DDoS attacks originated from grid itself.

Section 5.1 introduces the models of the attacks that might be possible in grid systems, followed by the introduction of additional functions of the accountability agents for detection and protection in Section 5.2. Section 5.3 describes how the agents detect the attacks by using the accountability data.

5.1. Distributed Denial of Service Attacks Involving the Grid

Karig et al. [52] classify remote DDoS attacks into five different types: network device level attacks, operating system (OS) level attacks, application level attacks, data flooding, and attacks that exploit protocol features. Although DDoS attacks involving the grid can be of any type among these five types, we focus on the application level attacks

and data flooding attacks, so as to better understand the mechanisms of detection and protection from the DDoS attacks.

Depending on the location of target, we divide the DDoS attacks by grids into the following two types.

5.1.1. Attacks to a Server Located Inside the Grid

Attacks of this type, shown in Figure 5.1, target critical objects of the grids. These attacks make particular services inoperable by using grid resources to exhaust grid objects [52]. For example, if a centralized grid scheduler fails due to an attack, the whole system can fail [53]. Likewise, if the web services running at the gatekeeper are out of service, user requests through that gatekeeper can be denied.

A scheduler in a HN can become unavailable due to heavy load. If a large number of jobs need to be resubmitted to a scheduler within a very short time interval from the CNs located at different clusters, a queue that stores jobs according to the submission order cannot properly process all the submitted jobs. Therefore, due to the limited capacity of the queue, jobs continuously submitted by a malicious user can saturate all available queue space, resulting in legitimate users' jobs to be dropped or suspended. In this scenario, if the attacker sends the jobs at a speed faster than the job processing speed, the queue will be filled with the attacker's resubmitted jobs. Until all of the attacker's queued jobs are complete and exit from the queue, legitimate jobs cannot further proceed. By sending the same number of jobs to the same HN over time, the attacker can totally use up the queue, thereby making it impossible for legitimate users to submit their jobs. For example, if a legally submitted GRAM [12] job waits to be queued for a very long

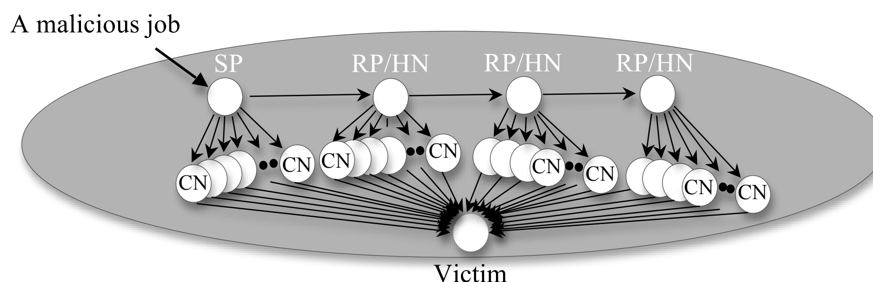


Fig. 5.1 Distributed Attacks on a Server Located Inside the Grid

time, the job manager terminates the job by cancelling the operation when the operation is timed out. Even if a queue does not become saturated, when the attacker continuously sends jobs from the CNs, a legitimate user will have to wait until the attacker's queued jobs are processed.

5.1.2. Attacks to a Server Located Outside the Grid

Grid resources can be exploited to make it impossible for any user, within and outside the grid, to connect to a remote server. Attackers can target high-profile web servers of banks, credit payment gateways, or mission-critical governmental hosts by executing code or by invoking shell programs that contain applications in order to generate network traffic toward the victim node. This out-bounding network traffic can consume the entire network bandwidth in a short time, thus making the connections unavailable to legitimate users (Figure 5.2). If the malicious code in each CN concurrently and continuously sends packets or generates heavy loads of page requests to the victim, the victim's server continues to be out-of-service until the job execution is completed.

5.2. Tasks of Accountability Agents for DDoS Attacks

In order for the accountability agents to be able to thwart possible DDoS attacks, the agents' capabilities need to include additional capabilities, such as detecting anomalies, issuing alarms, and taking proper responsive actions.

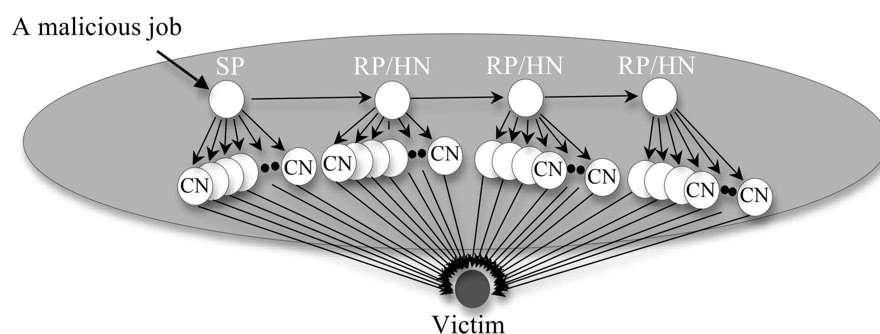


Fig. 5.2 Distributed Attacks on a Server Located Outside the Grid

Detecting anomalies. In order to detect possible attacks the agents gather information from monitoring tools, such as process accounting tools [54], or bandwidth monitors [55], according to the specific object being monitored. When no dedicated tool monitoring a specific grid object, such as a queue in the HN, the set of files opened by a grid portal, and so forth, is deployed, the agents directly collect consumption data about the object through the logs generated by the processes that utilize the object or through system commands such as '*lsof*' and '*strace*' [56]. The agents employ a statistics-based [57] or entropy-based approach [58] to detect usage anomalies using the collected consumption data about the grid object. Upon detection of anomalies, the agents further investigate signs of attack by raising alarms.

Raising alarms. To notify of a possible attack, agents coordinate with each other by means of alarms. An alarm contains not only the warning itself, but also job information, such as handle, job id, process id, alarm-issuer's identity, and information about the possible target. Based on the detection stage and the likelihood of the attack, the alarm is classified as *light, moderate,* or *critical*. The alarm starts from a light level and then escalates to a critical level via a moderate level. The agent located at the node where the signs of an attack are first detected raises a *light* alarm to the agent in the predecessor node in the job-graph. Upon further detection of anomalies, a *moderate* alarm is promulgated. When many agents at CNs send a *light* alarm to a HN within a short time, the agent at the HN checks the information sends a *light* alarm for further detection and then sends a *moderate* alarm to the RP if it confirms that the multiple indications are in fact a sign of the attack. The existence of these indications depends on the job relationship that in our approach is modeled by the job-graph (As an example consider the job-graph in Figure 2.5). In such job-graph, the CN2 of job relationship (Figure 2.5-f) has only one adjacency list (i.e., one outgoing edge from CN2); thus the agent in CN2 will have only one *light* alarm from the successor node. Data sent in one light alarm does not provide further indicators of attacks. In this case the agent just transmits the information received in the alarm to the direct predecessor node in the job-graph. The destination where to send the alarm is indicated in the cover-records that contain the job relationship. If the sub jobs used for attacks from distributed CNs have a common handle

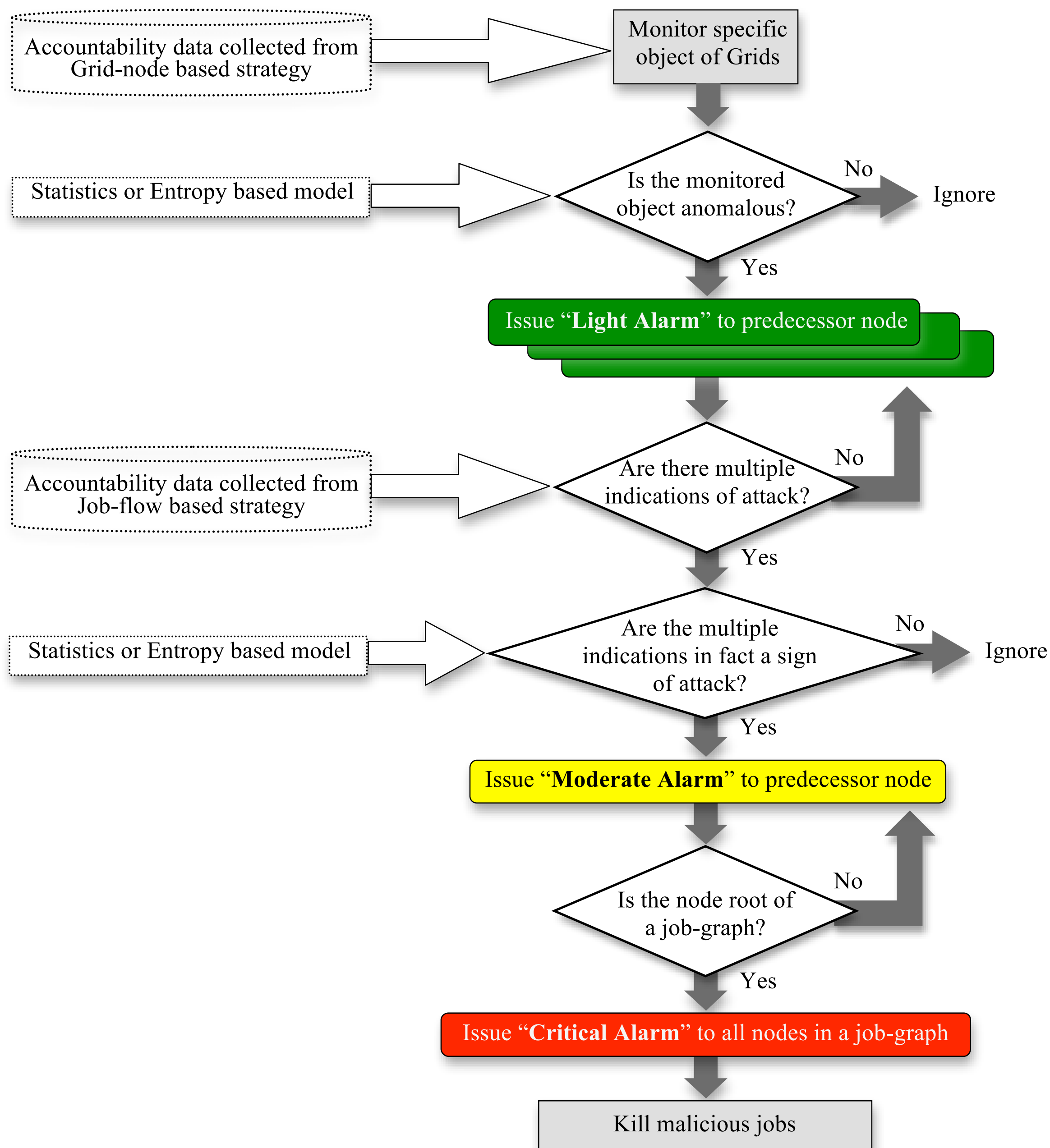


Figure 5.3 Steps for Issuing Alarms

or are originated from the same location, this can be evidence indicating possible attacks because it is very atypical behavior in grid. The RP sends a *moderate* alarm received from another RP, as a carrier of the alarm to the agent in the root node when it is not a root node. When the agent in the root node is informed of the reported *moderate* alarm, it triggers a *critical* alarm to all nodes in the job-graph. Once a node is notified of the

Table 5.1 Classification of Alarms

Type	Severity	Sender	Receiver	Action when received
Light	Low	First node that detects sign (CN/HN/RP)	HN/RP/SP	Ignore/Issue another *Light*/Elevate to *Moderate*
Moderate	Medium	Intermediate node (HN/RP)	RP/SP	Issue another *Moderate*/Issue *Critical* to all node
Critical	High	Entry node (RP/SP)	All nodes	Kill Process

critical alarm, the agent in that node takes responsive actions based on the information, such as job id and process id, sent in the *critical* alarm. The issuing steps and classification of alarms are summarized in Figure 5.3 and Table 5.1, respectively.

Taking responsive actions. When the agents are notified of the attack by a *critical* alarm, intuitively, the first action to take is to terminate the malicious jobs, by sending a kill signal to the specified processes running on behalf of the malicious jobs or deleting jobs waiting in a queue. For example, in LINUX/UNIX, the '*kill -1 process_id*' operation for removing process even with child processes in memory or '*qdel*' operation for PBS [20] scheduler can be used. The agent identifies the process to kill by the process id and the jobs to delete by the job id sent in the *critical* alarm. As next step, they block or cancel the sub-jobs that are assigned to other nodes existing in a job-graph and have not yet been activated or executed since such sub-jobs represent a potential danger. The agents in potentially dangerous nodes check the jobs at the first stage of job status (i.e., pending for example) with the handle sent in the *critical* alarm for a given amount of time to block the attacks. Finally the identified patterns of the performed attacks are recorded and analyzed for future prevention to assist with decisions such as determining the threshold values used for alarms.

A detailed example of execution of these tasks is described in the next Section.

5.3. Detection Strategies

5.3.1. Detection at the Victim Node

To detect patterns indicative of attacks, we use the accountability data collected according to the strategies introduced in Chapter 2. From the data obtained in the grid-node based strategy, the agent is able to detect anomalies concerning the usage of the grid objects including grid resources. However the techniques used to detect such anomalies are often not accurate and can result in a high rate of false detection, especially when applied to complex systems such as a grid. For example, if many grid users submit their jobs to a certain grid gatekeeper or to a certain queue of a cluster by chance, for a short time, it is very difficult for existing intrusion detection tools to accurately distinguish a DDoS attack resulting from intentional malicious submissions from a peak in the server resulting from legal submissions. If multiple attack servers operate in coordinated fashion against the victim, it is almost impossible to detect such an attack [59].

Under our approach, upon detection of a potential DDoS attack, the agents do not immediately consider it as an attack attempt. By using the accountability data concerning the job's flow collected from the job-flow based strategy, the agents trace back the job transmission path to send the detected information in an alarm message. By combining accountability data collected from multiple nodes, our approach is able to gather more clear signs of attacks. The agent in the upper node applies the existing anomaly analysis methodologies (for example a statistical model) to data obtained by the children nodes again.

When monitoring grid objects including grid resources, the agent checks them periodically at the end of a given time interval called a *sliding time window* or *time window*. The size of the time window depends on the characteristics of objects. We use a queue as an example of critical grid objects to show the detection and protection mechanism at the victim node.

In order to set an adequate size for the sliding time window to monitor a queue, the following factors should be considered: the average number of jobs entering the queue per time unit, the average processing time per job, the available queue size, and the

number of CNs where the actual execution is performed. The difference between an outgoing and incoming job's flow per time unit is the rate of the remaining jobs per time unit. By dividing the maximum queue size by the calculated remaining rate of jobs in the queue, we can obtain the time required to fill the queue to capacity. Thus, the time window (*TW*) is calculated as follows.

$$TW = \frac{J_{\max}}{(\frac{J_{in}}{t}) - \frac{c}{(\frac{T}{j_{out}})}} \tag{5.1}$$

where J_{in} / t is the average number of jobs entering the queue per time unit, t; T / j_{out} is the average processing time per job; J_{max} is the queue size; and c is the cluster size, which is the number of CNs for the queue.

If the denominator in Equation 5.1 is less than or equal to 0, then there are no remaining jobs in the queue because the rate according to which jobs are processed is higher than or equal to the rate according to which the jobs enter the queue. Since such case is not indicative of a malicious action, we ignore it and assume that the denominator is always greater than 0.

To saturate a queue, the attacker will try to increase both the number of nodes from which to submit jobs and the execution time of jobs, which can be modeled in Equation 5.1 by increasing J_{in} / t and T / j_{out}. From the estimation of the increase in J_{in} / t, and T / j_{out} with the known size of the queue and the cluster, the size of the time window *TW* can be obtained. If *TW* becomes small, the agent will keep track of the queue usage more often. Clearly, a small window size implies higher costs in terms of resource consumption for monitoring purposes.

Table 5.2-b shows the number of queued jobs with job identifiers at the end of a given time interval called *time window*. We use a queue as an example of critical grid objects to show the detection and protection mechanism starting from the victim node. The assumptions in this example are as follows: a time window slides every 10 seconds; statistically filling 90% of the queue is considered abnormal; and the queue size is 25. Under such assumptions, when the sliding time window is at 10:03:18, a number of

Table 5.2 a. Collection of Handles for Each Job Id Based on Cover-records Created in the Job-flow Based Strategy (Left table); b. Data Collected According to the Grid-node Based Strategy (Right table). Hndl=Handle, Q'd=Number of Queued Jobs, Queue Size=25

Hndl	Job Id	Hndl	Job Id	Hndl	Job Id
ske	job1	abc	job9	abc	job17
ske	job2	abc	job10	abc	job18
wai	job3	abc	job11	abc	job19
wai	job4	abc	job12	abc	job20
abc	job5	abc	job13	abc	job21
abc	job6	abc	job14	abc	job22
abc	job7	abc	job15	abc	job23
abc	job8	abc	job16	abc	job24

Job Id	Q'd	Time
job1~2	2	10:03:09
job3	1	10:03:13
job4~5	2	10:03:18
job6~9	4	10:03:20
job10~13	4	10:03:22
job14~17	4	10:03:24
job18~20	3	10:03:26
job21~24	4	10:03:28

monitored queued jobs down to 10:03:09 appears to be legitimate because the sum is 5, thus only 20% (i.e., 5 out of 25) of the queue is filled. At the end of next time window (i.e., at 10:03:28), the number of queued jobs looks abnormal because the sum of the remaining jobs in the queue until 10:03:28 is 24 (assuming jobs 1 through 5 are still in the queue); thus 96% (i.e., 24 out of 25) of the queue has been filled at 10:03:28 for a short time (i.e., for two time windows). However, even though the status of the monitored queue is considered abnormal, this anomaly does not immediately trigger a defensive action against a potential DDoS attack, but it simply raises a *light* alarm to the direct predecessor nodes in the job-graph. If the victim node in Figure 5.1 is a HN which is a scheduling node, the agent in the HN by referring to the job-relation in the cover-record finds out that the job is submitted from CNs and then sends a *light* alarm to these CNs. The *light* alarm includes collected accountability data, such as *{job id, handle, and timestamp}* from the cover-record, and *{executable name, process id}* from the resource usage record. Because a CN in Figure 5.1, for example, has only one adjacency list in the graph, the agent in such CN just needs to send the received data to the RP/HN in the job-relation of its cover-record in the *light* alarm. The agent in the HN counts, using Table 5.2-a, which combines the matching handle and job-id sent by the CNs, how many job-ids are associated with the handle. If within the monitored time interval the same handle

is associated with a number of jobs within the threshold, the attack is not considered as a DDoS and the *light* alarm is ignored. If it is out of the threshold, the agent located at the victim's node raises a *moderate* alarm to the agent located at the direct predecessor node. In this example, Table 5.2-a reports 20 jobs for the handle, *'abc'* and 4 jobs with different handles (*'ske'*, *'wai')*. Hence, 20 out of 24 (83%) of the jobs are multiple submissions of the same job. In a grid computing system where a job is split into many sub-jobs to be run in parallel at multiple CNs, multiple sub-jobs resubmitted to a scheduler are considered suspicious. As a result, the *moderate* alarm is sent to SP or another RP in Figure 5.1.

If the malicious job flows through multiple RPs in order to take advantage of more computing resources from different domain as shown in Figure 5.1, the *moderate* alarm will be relayed by each RP and finally will arrive at the root node (SP) of the job-graph. When the agent in the root node receives a *moderate* alarm, it triggers a *critical* alarm to all nodes in the job-graph. Upon receiving a *critical* alarm, the agent increases the priority of jobs identified as legal or deletes malicious jobs in the queue. By exchanging the accountability data in real-time, the agents can quickly identify the nodes where the signs of attacks are not yet actually detected, and timely terminate sub jobs that may potentially perform malicious actions before launching the attacks (i.e., at pending status or before submission).

5.3.2. Detection at the Source Node

During an attack against servers located outside the grid (see Section 5.1.2), the agents do not have any control over the victim's server. Thus, it is almost impossible to detect the source of attacks and stop the ongoing attacks in the victim's server outside the grid, since the agents do not reside in such victim's server. In order to address this issue, an approach to detect and stop the malicious activities at the attacking nodes is required. In an accountable grid computing system, the agents have the right to collect data at each node and the ability to monitor the job activities across the different domains. By analyzing the data collected according to the grid-node based strategy, we can obtain useful indications to detect a DDoS attack based on the following observations: *1) the*

normal behavior of the CNs is to execute jobs at computational resources; 2) If the job executions induce heavy out-bounding network transactions in every CN, they can be considered as abnormal as such behavior is very atypical of jobs executed in grids. When the destinations of most network transactions of a job have the same address, this job can be considered very suspicious and is most likely launching a DDoS attack against the server or website located at that address.

Monitoring the processes created by the job running in a CN can help in analyzing the behavior of the CN. If the job is scheduled by a PBS and placed into execution by a PBS Machine Oriented Mini-server (pbs_mom) [Staples, 2006], monitoring should be performed by first tracing the pbs_mom. Because a pbs_mom places jobs into execution mode, monitors the job's usage, and notifies the server when the job completes, tracing the daemon running for the pbs_mom provides enough information about the job, including system calls, name of script, name of executables, each with the process id. If the job is scheduled by a Condor-G scheduler [14], the condor_startd [60] daemon is the right process to start monitoring in order to trace the currently running jobs. The profiled process information can also be used to check whether a job results in heavy out-bounding network transactions (see *observation 2)*. We monitor the files and especially the network files opened by the program executing in the CN. Such network open files show the source and destination address, each with process id and application name. By combining the profiled process id or the name of the executable with the process of interest, we can obtain destination information bounded outside the grid.

When the agent in a CN applies existing anomaly detection models, such as entropy or statistical models, to the obtained destination information and detects that many packets are sent outside the grid as a result of executing the submitted job, the agent accesses the job flow information recorded in the job-relation of the cover-record. From this information, the agent at the CN determines where the jobs that caused the anomalous behavior originated and sends a *light* alarm to the direct predecessor's agent, for example RP2 or HN2 in Figure 2.5-d. The agent in HN2 determines from the alarms reported by the CNs how many sub-jobs have the same destination for their network transactions again in the applied model. Based on a comparison with the threshold

defined according to the statistical model, the agent in HN2 decides whether it will issue a *moderate* alarm to the upper node (i.e. RP2 or SP1 in Figure 2.5-c). This threshold is required in order to capture the fact that some legitimate sub-jobs can have out-bounding network transactions, such as sending outcome files to other nodes, but not all of them send the outcomes or data packets to the same server at the same time. Without an intentional purpose, it is unlikely that the same destination will receive several packets within a short time from the CNs. Even if the attacker generates packets for various destinations to hide the attack, at least a certain number of requests to the same target must be issued in order to saturate the bandwidth. This large number of requests represents a possible symptom of a DDoS attack.

Once the sub-jobs with malicious code have been delegated through multiple RPs, as shown in Figure 5.2, the agent in each RP reports a *moderate* alarm to its direct predecessor node (i.e. the delegating RP). Since each agent has a partial view of the job-graph, the accountability system can trace the original job. The agent at the root node finally issues a *critical* alarm when it receives *moderate* alarm to all nodes in the job-graph, and all queued and running sub-jobs are then terminated. The detection and the decision are made very quickly to shorten as much as possible the out-of-service time.

6. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

A prototype of the accountability system has been implemented and evaluated on an emulated grid test-bed. Our test-bed consists of a hundred nodes, allocated using Emulab [1] as SPs and RPs and clusters. Each RP is connected to a scheduler, which has multiple compute nodes comprising a cluster. The GT4 [18] is installed at the SP and RPs. We placed the agents according to the two strategies introduced in the previous Chapter; agents are placed at the SP, the gatekeepers of RPs, the scheduler and the compute nodes. We used the PBS [20] for job scheduling. Accountability data are stored according to a distributed strategy in which each agent has its own local database system. We used postgresql for its good performance [61].

6.1. Implementations of Agents

Each agent is composed of a library of functions, the most important of which are: the function that retrieves data from GT4 [18] and PBS; the function that updates the database; the function that supports the interactions with the other nodes according to a client/server mechanism; the function that manages the accountability policies; the function that applies existing anomaly detection tools for collected data; and the function that protects the system from attacks. Below we highlight the most interesting implementation issues we had to face during deployment.

Dealing with Grid middleware and Schedulers. One critical issue is whether existing monitoring approaches and log files available at job schedulers and gatekeepers are sufficient to support our accountability approach. An obvious source for job-related information is the log file generated by the Globus container. However, we found that the Globus log files alone did not provide sufficient job information at the level of detail we

require. For example, when a composite job is submitted, information about the sub-jobs, such as sub-job id and destinations, is not recorded in the log files, although this information exists as properties of sub-jobs. To address this problem, we extended Globus so that when a GRAM [12] job is submitted, accountability data, which at time of submission consists of the initial user's handle and job id and associated resources that the user intends to submit, is recorded in the agent's database.

We also modified Globus to support the communication of job state changes to accountability agents. During its normal execution, a Globus GRAM job can be in different states, including *'StageIn', 'Pending', 'Active', 'Done'*. With our modification, the accountability agent is notified by an instance of *StateMachine()* (a Globus routine) whenever the job state changes. We parsed PBS log files at the head nodes for job scheduler. These log files provide additional information about jobs and grid resources, including the job flow information (e.g., name of the compute node where the sub-jobs are assigned, and names of the sub-jobs). Such information is obtained and passed to the agent when sub jobs are assigned to the compute nodes by the scheduler. Because the PBS job id is also used in the Gobus log, our agents can uniquely map a GRAM job id to a PBS job id – this linkage provides the necessary information to create a job-relation graph. Notice that the agent has no way to connect the two identifiers until the agent in the predecessor node pushes such information.

For simplicity, in the implementation, we simulated the GridShib handle with the handle uniquely generated by Globus for each job submission. Each cover-record thus maintains a unique identifier given by the unique user handle and the job id.

Primitives. We embedded fine-grained monitoring primitives, encoded using Java, in few Globus routines. Specifically, we capture the information necessary to create and maintain the graph-based logging, such as handle, job id, sub job id, the destinations where sub-jobs are assigned, and timestamp. We also extended the *StateMachine.java* to include certain agent's information and data, especially the data specified by the policies. This extension allows the state machine to pass such information to the agent. The routine *ManagedMultiJobResources* was modified for collecting composite jobs-data. *ManagedMultiJobResources* creates sub-jobs, collects data upon state change, and pushes

it to the agent. The specific information actually collected and stored in the database is filtered based on the policies.

Definition and evaluation of policies. An interesting challenge was how to implement local and shared policies. To properly enforce such policies, outputs from several primitives must be gathered. Moreover, when enforcing a shared policy, each local agent must coordinate with other agents. We implemented such policies as XML files, to be created by administrators off line and then stored in the local directory of the agents. Shared policies are evaluated whenever a job state change occurs. Precisely at SP/RP policies are evaluated when a notification from Globus is received about a change in the job status. At other locations, the job state change always triggers a policy lookup process, to search for potential policies that need to be applied. When evaluated at first, policy files are parsed into database tables only once to save the file accessing expense, and then the tables are accessed to identify the data that need to be collected. When available, such data is first locally stored. Then, specific agents' functions are executed to send/receive the data specified by the policies. For example, when a job state changes from *StageIn* to *Pending*, basic accountability data is gathered. Additionally, when the job moves to the *Active* state, an agent interacts with other agents located at nodes where the job is assigned in order to send sub-job information (as specified by the action expressions) gathered at *Pending* state to agents located at successor nodes of the job-relation graph.

Unique Identification of sub-jobs at the compute nodes. In the case of PBS, when a job is split and processed in parallel, PBS does not assign any new job id to it. Thus, in order to be able to determine all the nodes at which portions of the job are allocated, the local agent needs to maintain additional information and locally generate unique sub-job ids. Specifically, the agent collects mapping information to find resource information associated with the job. In case of parallelized sub-jobs (e.g., the node has several computational units) the agent maintains rank and node information in the PBS log file that allows to distinguish job portions at the finest level of granularity. Finally, in case of loops, that is, when the same job is assigned multiple times at the same compute node, the timestamp helps in differentiating the various job records.

In order to enable communication between agents, the client/server model is used. The agent acting as a client is implemented by a thread so to handle multiple concurrent executions. For example, the agent at head node of clusters works as a multi-threaded client when it contacts compute nodes to provide the accountability data. Threads are also employed for the PBS logging modules. Implementing the agent using threads makes the agents monitoring tasks completely transparent to the ordinary job execution. This level of parallelization results in a very efficient and light-weight approach, as shown by our experimental results.

6.2. Configuration of Experiments in the Emulab Test-bed

As already introduced, the experiments have been performed by using the Emulab test-bed. The machines used at the various nodes for the experiments are of the following types: pc600; pc850 hosts which are 600MHz Intel Pentium 3; pc3000, 3GHz Intel 64-bit

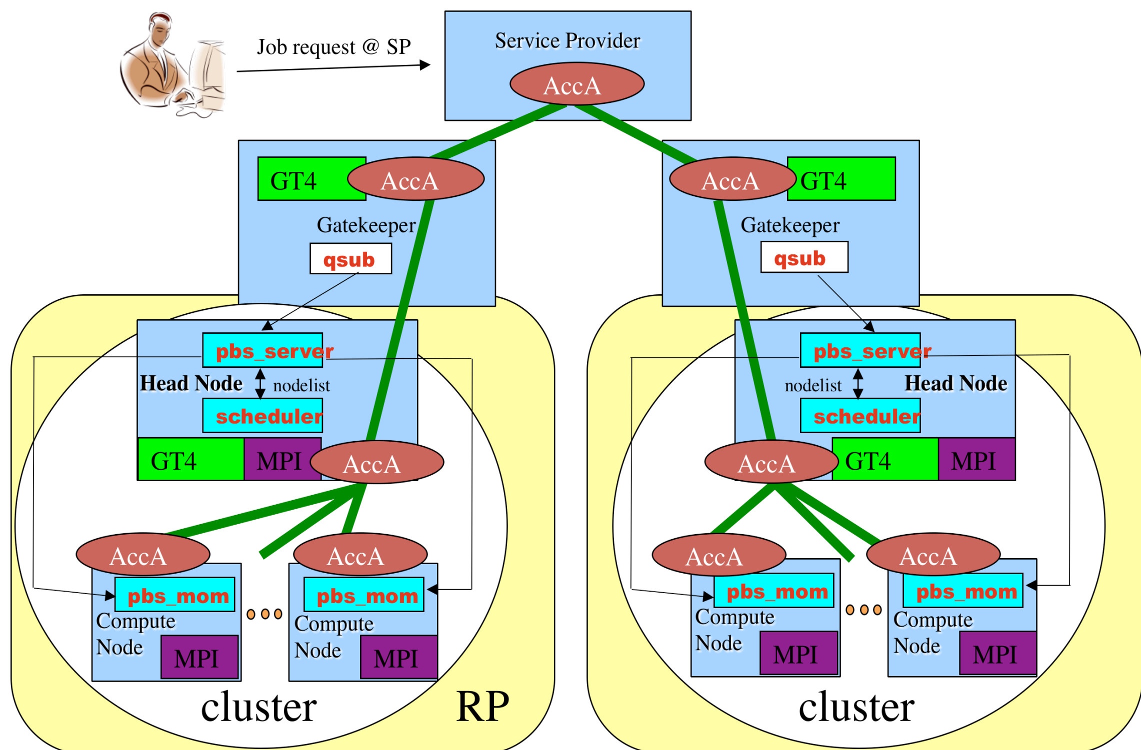


Figure 6.1 One Use-case of Job Submission

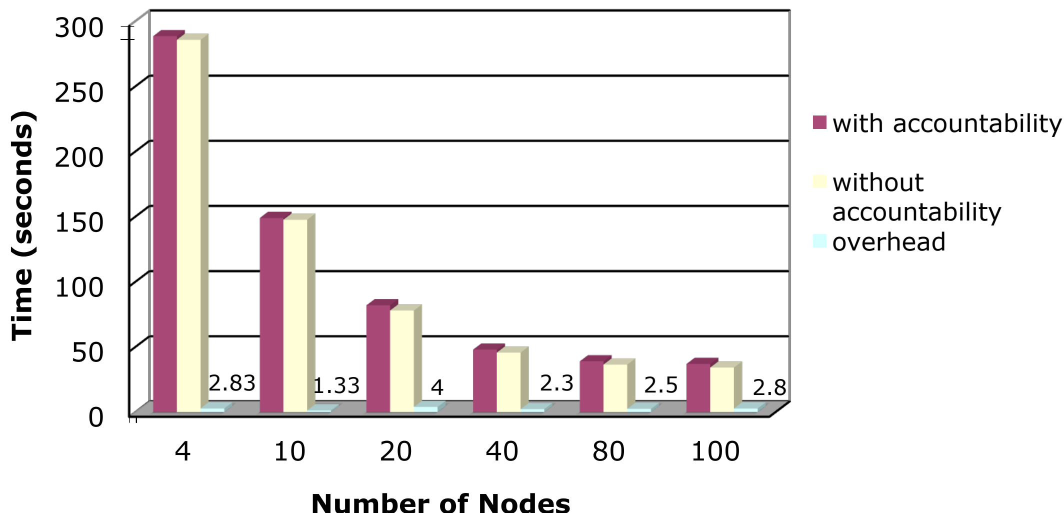
Xeon; pc2400w and pc2400c2, 2.4GHz Intel Core 2 Duo; pc2000, 2GHz Intel Pentium 4; pc3000w, 3GHz Intel Pentium 4. To begin the Emulab experiments (it is called *swap-in*), a number of free PCs are selected based on the machine node types and assigned to the experiment according to the number specified by the Network Simulation (NS) script. In our experiment, the NS script specifies the number of nodes by distinguishing them between head nodes and compute nodes. In the experiments we use the Emulab Operating System Image created in Fedora 2.6.23.15-13. This image contains installations of OS and GT4, the basic software required by GT4, postgresql-8.2, PBS (torque-2.1.8), compilers (gcc-4.1.2, java-1.5.0_14, mpich2-1.0.5p4, etc), and the basic configurations to install software applicable to all nodes. When starting the experiment, the OS image is loaded by reading the NS script where the node-dependent tasks are also specified. Examples of such tasks are: configuring the GT4 by issuing certificates for the hosts, Globus container, and user; installing additional software required to specific nodes; and configuring the installed software depending on the node's roles such as RP, HN, and CN. At the last step, the Globus container, PBS, and agent are started to run. By changing the NS script, we generated different grid topologies. One basic sample configuration in a topology is introduced at Figure 6.1.

6.3. Experiments

The goal of our experiments is to assess the scalability of our approach and the performance of the protection system. Resources in terms of grid nodes scale by adding compute nodes at the same administrative or by federating other institutions. Following sections show the scalability assessment in two approaches and some policy evaluations followed by performance evaluation for detection and protection against DDoS attacks.

6.3.1. Scalability with respect to the number of computing nodes

In the first experiment, we measured the job execution response time for increasing values in the number of compute nodes [see Figure 6.2]. We also evaluate the scalability with respect to the applications size by running different applications that have different execution times



Graph 6.1 Overall Response Time for Job Completion for Different Number of Nodes.

Graph 6.1 shows the response time for a job submission; the response time is computed as the difference between the time at which the user receives the result and the time at which a user submits the job. We measured the overall time for conducting these operations in a grid with and without the accountability system in place. The same job is used for all different cases of the experiment. Such job computes prime numbers between 0 and 100 millions and returns the highest prime number and the total number of prime numbers within a certain range specified by the user. The job is split onto a number of

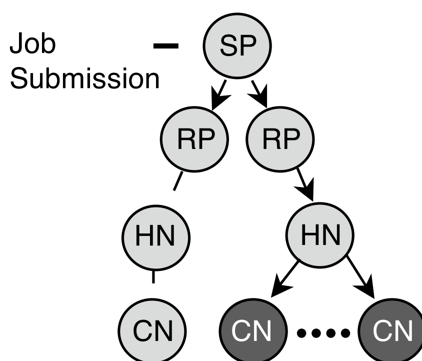
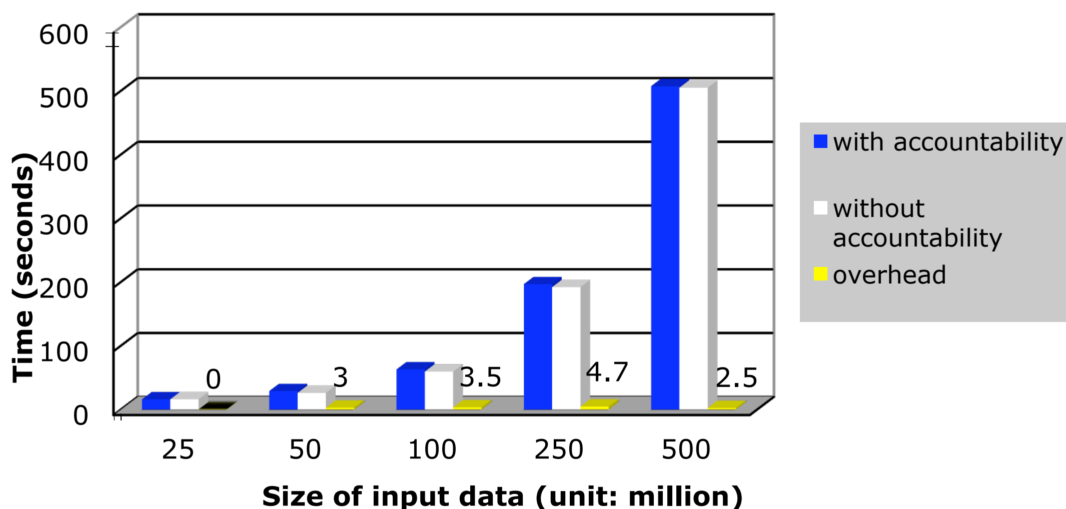


Figure 6.2 Job Submission to Multiple Compute Nodes



Graph 6.2 Overall Response Time for Job Completion for Different Execution Time.

compute nodes for parallel execution. As shown by Graph 6.1, the overhead introduced by the accountability system is negligible.

Graph 6.2 shows the response time for varying runtimes of the applications. We make the application used in experiment run for input data of different values required. We measured the execution time for a grid composed by 40 nodes, and compared the execution time in the case in which the accountability system is in place and in the case in which it is not. The number of nodes does not change (it was 40 in all cases).
The blue bars (in graph 6.1), and yellow bars (representing the differences between the times reported by blue bars and the times reported by the white bars at graph 6.2) in both graphs indicate that the overhead introduced by the accountability system is constant (between 2 and 3 seconds) with respect to the number of nodes in the grid and the size of the applications. As shown in graph 6.1, even though the job involves 100 nodes, the accountability system does not impact the performance because our implementation strategy, according to whole time-consuming functions work asynchronously with respect to the GT4 and PBS. Graph 6.2 shows that the overhead for the accountability system is not dependent from the application execution times, and is negligible, especially when running long jobs. In conclusion, this experiment clearly demonstrates that our

accountability system is lightweight and does not interfere with the ordinary computation and activities of a grid computing system.

6.3.2. Scalability with respect to the number of Resource Providers

In this experiment, a job is repeatedly submitted to multiple RPs under the assumption that there is a gatekeeper at each RP. This scenario can occur when a RP does

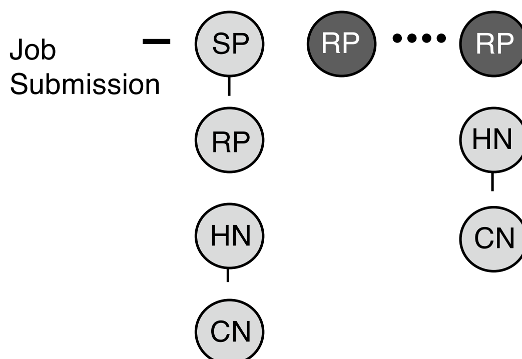


Figure 6.3 Job Submission Across Multiple RPs

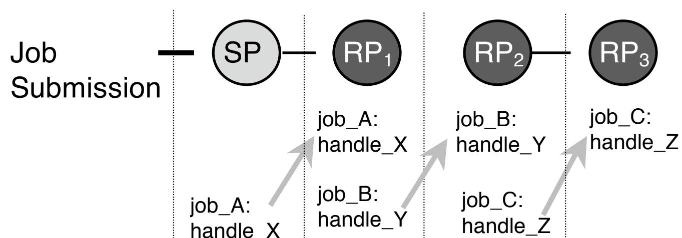


Figure 6.4 An Example of the Inconsistency in the Handle for Jobs Forwarded Through Multiple RPs

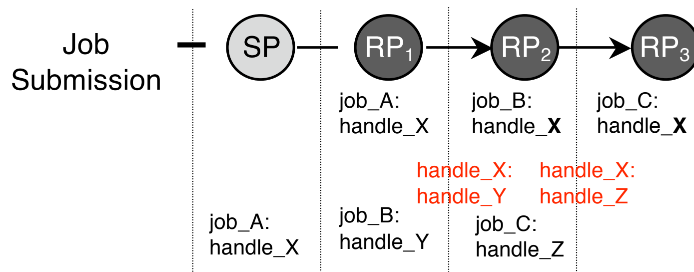


Figure 6.5 An Example of the Handle Consistency for Jobs Forwarded Across Multiple RPs

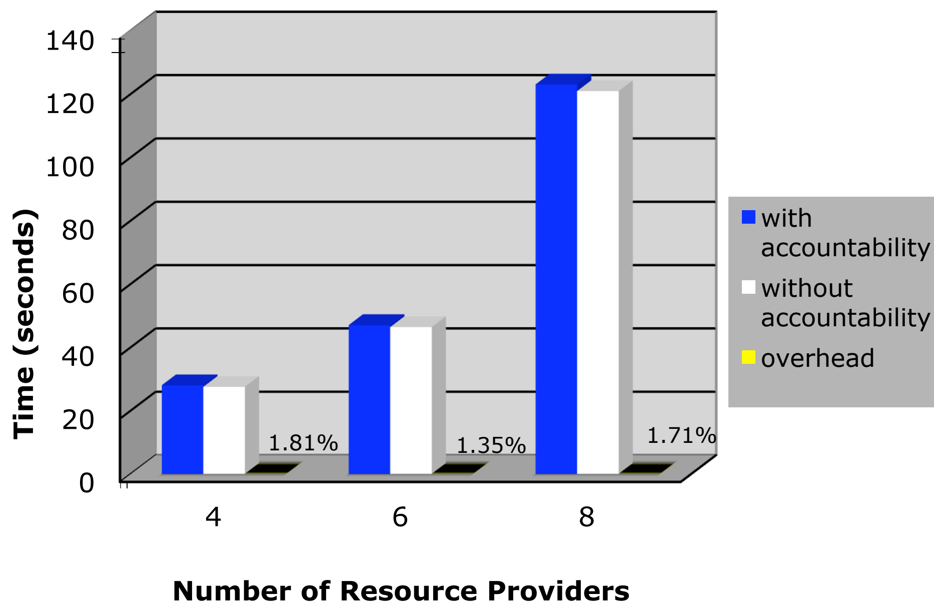
not have enough resources to perform the job execution, and thus it submits the job to another RP, or when a part of the job is submitted again to another RP. By increasing the number of RP nodes, we measured the job response time with and without the accountability agents. The job submission path assumed for this experiment is shown in Figure 6.3.

The number of compute nodes controlled by the final RP does not vary. Since GT4 does not schedule jobs between gatekeepers, we used a script to submit the first submitted job to another RP then repeating this submission, and then execute the job at the final RP. Users can actually submit a job in this way, by delegating the user credentials to multiple RPs. Thus, the scenario used in this experiment can happen in practice. When a job script is submitted and then re-submitted at a different RP, the job script execution and the job submission in the job script are considered as two independent operations by GT4. The OS does not provide any information about the relation of such executions back to GT4. Such lack of information introduces inconsistency in the handle generated at the entry point for a job. For example consider the example in Figure 6.4. In such example, though job_A at the SP is the same job as job_B, and job_C forwarded to RP_2, and RP_3 respectively, the job is considered a new job at each RP, resulting in three different handles. Figure 6.4 shows that job_A, which is executed at RP1, is submitted from SP with handle_X; thus handle_X is maintained at RP_1. The invoked job submission (job_B) from job_A is submitted to RP_2 with a different handle, handle_Y even though job_B is delegated from job_A and should have the same handle, handle_X. The handle information is again changed when the job is submitted again.

We addressed this issue by linking the various handles with the jobs they are associated with, at the job completion. For example, although job_A and job_B are considered different jobs by GT4, they are performed within one period of a job completion, which starts from the "Start" state to the "Completed" state. With this knowledge, we retrieve the pair of the previous handles and the new handle, which are then sent to the successor node, and construct the cover-records for job_A, and job_B at each RP. At the successor node (RP_2), the agent updates handle_Y to handle_X by

searching for handle_Y from the handle pair information. After the update at RP_2, the handle pair becomes handle_X:handle_Z. With this information, handle_Z is again updated to handle_X. We update every new generated handle for the same job with the original handle generated at entry point (see Figure 6.5).

Although the handle searching process may seem time consuming because of the many interactions with the database, the overhead introduced by the accountability system is negligible, like in the previous experiments, because of the thread-based implementation. When we tested a job submission on multiple RPs, we observed an overhead within 2% of the overall job response time as graph 6.3 shows. We expect similar results, also for larger number of RPs.



Graph 6.3. Response Time for Different Number of RPs

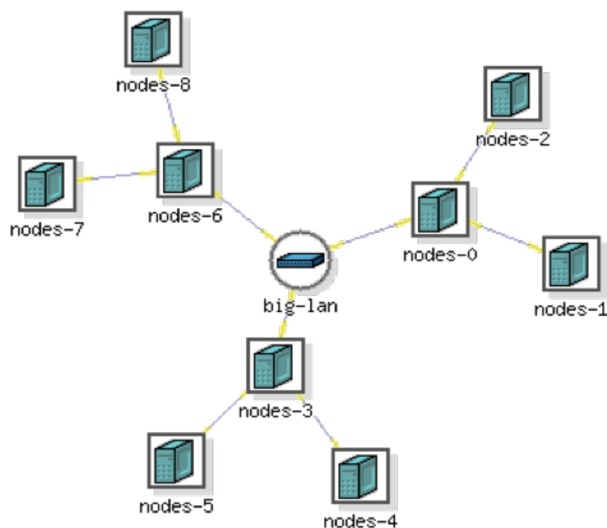
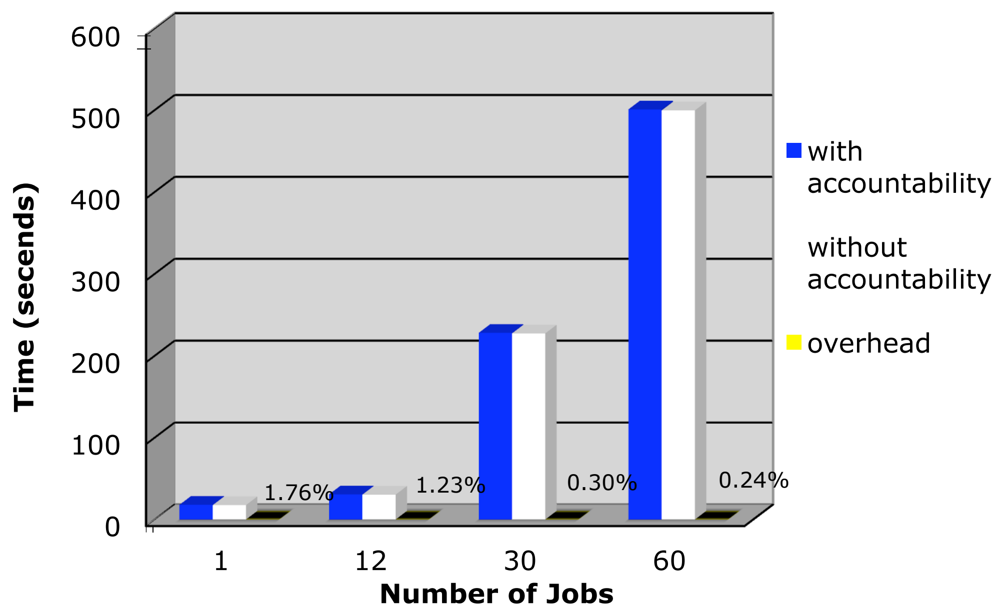


Figure 6.6 Topology for Experiment 6.3.3



Graph 6.4. Average Response Time for Multiple Job Submissions

6.3.3. Scalability across Multiple Domains

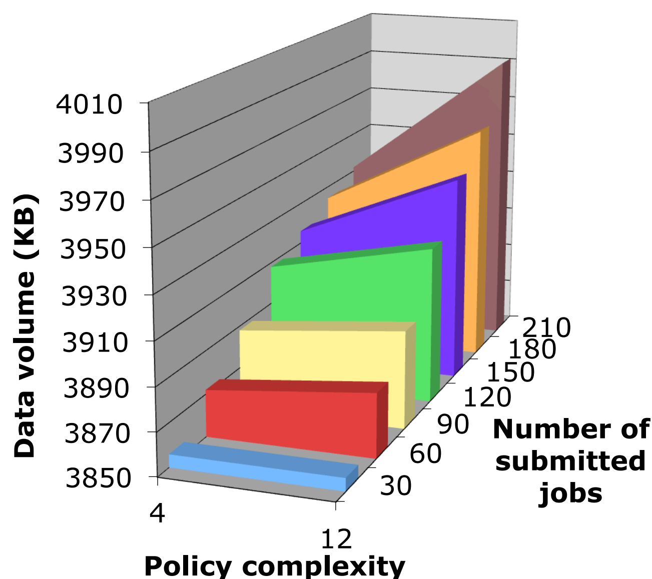
A crucial requirement is to assess to which extent our accountability system

degrades the performance of the grid computing system when multiple jobs by multiple users are submitted. In this experiment, by submitting several jobs from multiple locations at the same time, we measured the average job response time and evaluated the performance of our system.

The topology that we created in the Emulab test-bed for this experiment is shown in Figure 6.6. For this experiment, we considered the topology different from the one used in the previous experiments. In this grid networks nodes-0, nodes-3, and nodes-6 work both as a RP and HN. Each HN has two compute nodes. Multiple jobs are submitted to different RPs from each terminal nodes (i.e., nodes-1, nodes-2).
Our goal is to evaluate whether in case of multiple RPs involved in the multiple job submissions process at the same time, the impact of the accountability system is negligible as observed in the previous experiments. Multiple jobs are submitted from the compute nodes to two other RPs at the same time. We measured the average response time with and without accountability system. The results of this experiment, shown in Graph 6.4, confirm the results obtained by Experiment 1. The accountability system does not affect the performance of the grid system. The reason is that the agents at each location are implemented using multi-threads. The average time to process the shared policy at SP, RP, and HN, and the local policy at HN as represented at Figure 2.9, and to perform the actions required by the policy takes only around or less than 1% of the average job completion time. This percentile value decreases when more jobs are submitted.

6.3.4. Scalability with respect to the Data Volume

Most of the monitoring and accounting systems accumulate a huge amount of data. Data volume is the main concern for administrators. Since our accountability system is designed based on the notion of distributing the job-graph based-log, different portions of the accountability data required for constructing a job-graph are stored at different each agent's location, thus reducing the overall volume of data at a single location point. Furthermore, the use of accountability policies makes it possible for the administrators to save only selected accountability data. Using the policy language the administrators can



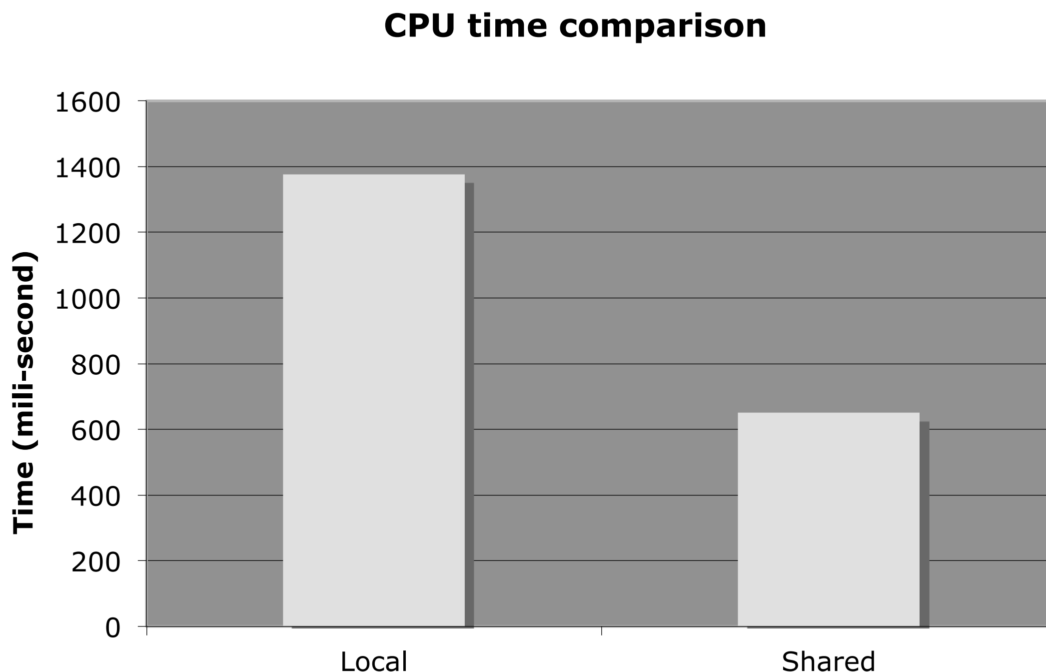
Graph 6.5 Data Volume for Different Policies

configure the accountability system so to record only some data. Therefore, if data volume is a concern for an administrator, the administrator can trade off accountability accuracy for performance. Graph 6.5 shows the relation between the different policies and data volume according to the number of submitted jobs. We employed different policies, with different complexities, and measured the data volume for a number of jobs ranging from 30 to 210. The policy complexity varies according to the number of elements of the policy ranging from 4 until 12.

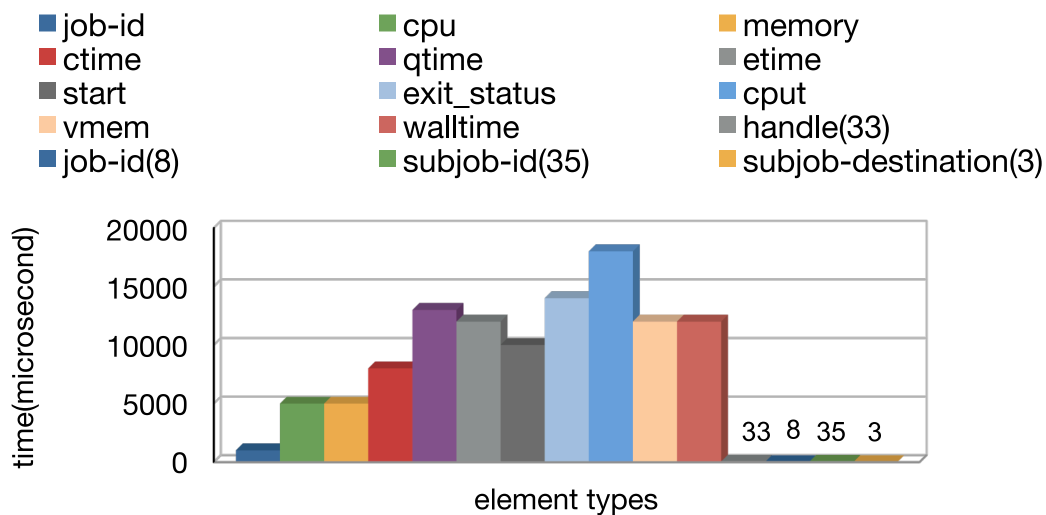
While a shared policy concerning job-flow based data is enforced upon a change of the job status, the local policies are applied when the agent at a node starts collecting data. The agent checks if the job is submitted and then scans the log files to obtain resource data based on the local policy. When several jobs are submitted, a complex local policy generates higher data volumes than a less complex policy as the result shows. However searching the optimal point between accuracy and storage volumes is a responsibility of resource administrators.

6.3.5. Evaluation of shared policies vs local policies

This experiment analyzes the policy processing time for local and shared policies



Graph 6.6 Comparison of Policy Process Time for Shared and Local



Graph 6.7 Search Time for Policy Elements

at a head node, where both local and shared policies are enforced. Policy processing includes reading the policy from the xml file at local computer, and collecting or searching elements specified at the policy. It does not include the time for database operations since we assumed that updating the database for the same list of fields does not make difference. Note that we used the same policy complexity for both sample policies even though they have different elements. This is enabled by counting an element that belongs to different action specifications as different element.

The average time of executing a job under the local policy is twice longer than the time taken by the shared policy as shown in Graph 6.6. This difference comes from the operations that have to be executed on log files. Such operation is required by the local policy. As we described in the Section 6.1, collecting accountability data directly from the grid middleware takes much shorter time than searching for the resource usage data from local file system. As a result, we conclude that using local policies is more expensive than using shared policies. This result is confirmed by next experiment, which analyzes the search time required for the policy elements specified in different policies (see Graph 6.7). The rightmost four elements (handle, job-id, sub-job-id, and sub-job-destination) are elements collected by the shared policies used in the experiment, while the others are of local policies. Searching one element of local policies takes from 1 to 18 milliseconds, while it takes only from 3 to 35 microseconds for elements of the shared policies. The majority of the time required by the shared policies (reported in Graph 6.6) is due to read operations on the policy file and to the construction of the data structures for storing the element values before obtaining the values of elements. This experimental result is important for advising guideline to administrators for the design of the accountability policies

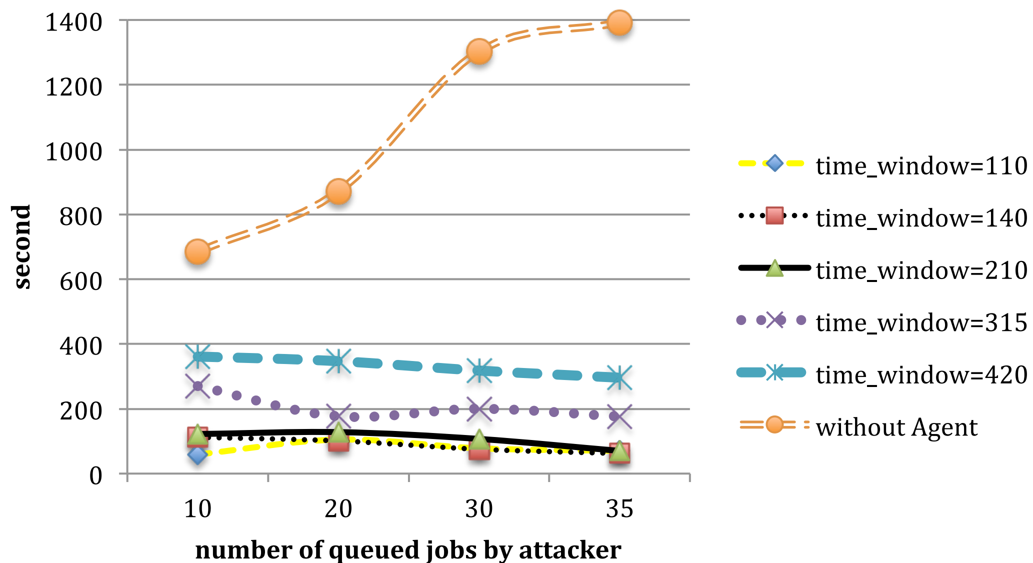
6.3.6. Detection and Protection from DDoS Attacks from the Victim-End with Time-Window 1

Our first experiment concerning DDoS attacks aimed at identifying a good time window size. A malicious job was submitted to an RP and then divided into 35 CNs to attack an HN located at the same grid by resubmitting jobs from 35 CNs to a queue in

that HN. While the attack was in progress, legitimate jobs were also submitted to the target HN. For each test run, we used 10, 20, 30, and 35 submitted jobs as the attacker's jobs. We assumed that the maximum size of the queue was 35[6]. We checked the queue size at every end of time window to see if the queue is filled to the degree that we consider anomalous. In our test-bed, an average of 19 jobs were queued every 100 seconds, and one job ran for about 78 seconds before exiting the queue. Since there were two CNs attached to an HN in this experiment, according to Equation 5.1, the time window size *TW* is calculated as

$$TW = \frac{35}{\left(\frac{19}{100} - \frac{2}{78}\right)} \cong 210$$

In this experiment, we modeled three PBS queues (i.e., *standby*, *standby-8*, and *tg_workq*) operating in the Teragrid [7] computing system at Purdue University to obtain practical threshold values. We checked the normal behaviour of the queues and determined that, on average, 25% to 44% of the queue was usually filled and only extraordinarily filled up to 81%. Based on this observation, we initially set the percentage



Graph 6.8 Normal Job's Wait Time for Different Time Windows

[6] In our emulated environment it is not actually feasible to saturate a queue.

of usage anomaly for the queue at 81%. We also observed that users submitted the same job multiple times to the same queue. However, such submissions did not fill the queue above 80%. From this second observation, we set the threshold of sub-jobs separated from a job with the same handle in a queue as 80%. Each time the queue size was checked, if the usage was over 81%, the agent that received this incident in upper node checked the job record to see whether the jobs were resubmitted. If the number of resubmitted jobs with the same handle was greater than 80%, then a *critical* alarm was issued to the HN from the agent in the entry node to kill the queued jobs submitted by the attacker. As a result, the legitimate jobs in the queue were not delayed and started running. The chart in Graph 6.8 shows the *wait time* of legitimate jobs until the job status went to the "active" state (i.e., the *running* phase) for different time window sizes. Legitimate jobs were submitted when the queue was filled with an attacker's job for 30%(10 jobs), 60%(20 jobs), 90%(30 jobs), and 100%(35 jobs). The top line denotes our baseline case, which is the wait time when the time window is too large or our system is not active. Since the attacker's jobs are queued before the legitimate jobs and run for a long time, the wait time increased as the number of the attacker's queued jobs increased. This means that, without the accountability agents or with a too large time window, the legitimate jobs submitted after the queue was filled with the attacker's jobs experienced a long wait to be queued and were thus unsubmitted.

The windows of size 210 and 110 appear to be the optimal. Windows larger than 210 seconds resulted in a loss of legitimate jobs after the queue was full for worst case, which is, when the attack starts together with the sliding time window. For windows that are 1.5 times and two times larger than 210, the wait time was much longer because malicious jobs were eliminated after the entire time window had passed. For windows smaller than 210 seconds, we did not observe much difference because the number of jobs in the queue do not exceed the threshold to detect the attack. This experiment shows that the legitimate jobs can be efficiently and effectively restored back to the normal execution with the help of the optimal time window obtained from Equation 5.1.

6.3.7. Detection and Protection from DDoS Attacks from the Victim-End with Time Window 2

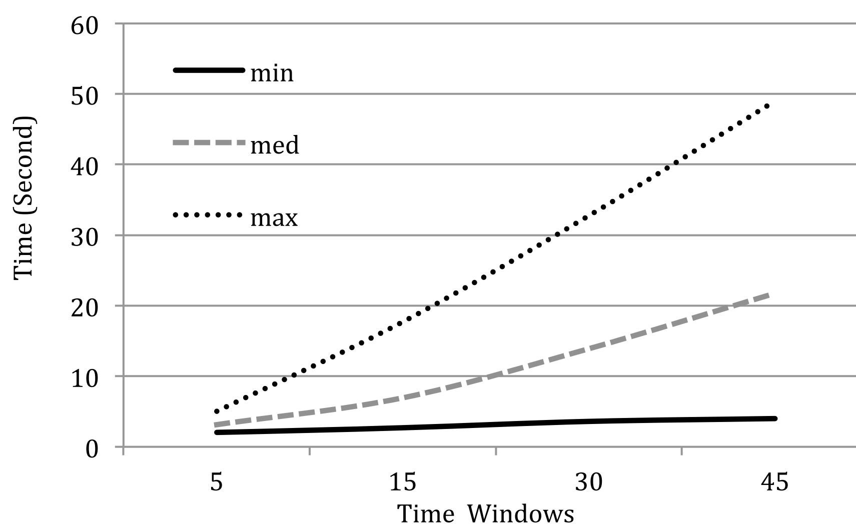
This experiment measured the time elapsed from the moment the attacks have been launched until when the attacks are removed. A malicious job was submitted to a SP and then divided into 6 RPs. Each RP has 1 HN and 3 CNs; thus 18 CNs from 6 RPs submitted the sub-jobs to a target queue of another HN. The job submitted by the CNs computes prime numbers between 0 and 250,000,000 and can be split into sub-jobs for parallel execution after having been compiled in message passing interface (MPI) [19]. In the experiment, we assume that the attacks are completely launched when all the 18 jobs are queued on the target HN and as a result, legal jobs cannot be queued or processed after 18 jobs are queued.

The first check for the queue usage is performed at the end of each time window. The threshold used in the first check was calculated from the PBS queues (i.e., *standby*, *standby-8*, and *tg_workq*), which were used for the Experiment in Section 6.3.6. The number of queued jobs are counted and recorded every 10 minutes for 1 month from the three queues. To detect anomalies in the queue usage, we used an entropy-based approach [58] because of its sensitivity and accuracy. The entropy [62] is the degree of uncertainty associated with a random variable. The entropy (H) of a discrete random variable X with possible values $\{x_1, x_2, \cdots, x_n\}$ and the normalized entropy (NE) are defined as

$$H(X) = -\sum_{i=1}^{n} P(x_i)\log_2 P(x_i), \qquad NE = \frac{H}{\log_2 n_0} \tag{6.1}$$

where $P(x_i)$ is the probability that X takes value x_i, and n_0 is the number of distinct values x_i.

By using entropy Equation 6.1, the minimum entropy was calculated for any range of the collected data from the 3 PBS queues. The calculated value was greater than 0.9, thus we referred to this value and chose a little lower value as the first threshold (i.e., 0.87) to consider obvious anomalies concerning resource consumption. The jobs submitted by the 18 CNs were quickly queued and the entropy was also calculated at the end of time window and compared with 0.87. The second check was performed at each RP to issue a *moderate* alarm to the root. We used the second threshold calculated in the



Graph 6.9 Detection and Recovery Time When the Attacks Are Completely Launched for Different Time-windows

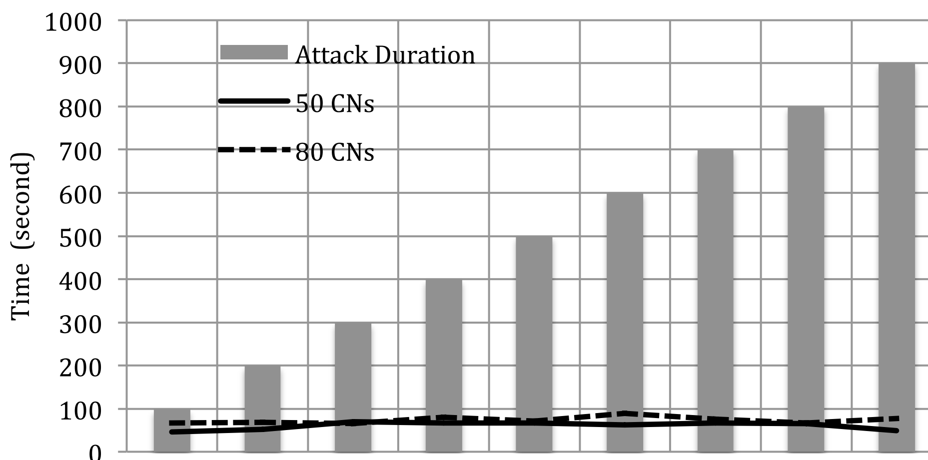
entropy-based model from data reporting how many identical jobs with different PBS job id are submitted to the same queue of the *steele* cluster [63] at Purdue University by the owner of the job to reflect real situations. The lowest entropy was 0.91; thus we chose 0.9 assuming that multiple submissions by a user to the same queue leading to entropy lower than 0.9 in entropy couldn't happen in a legal submission. After the second check, the time elapsed until all queued jobs are deleted by the *critical* alarm was also measured. The measured time can be classified as follows. *Time 1*: the time elapsed from the time the attack is completely launched to the first check; *Time 2*: the time elapsed from the time of the first check to the time of the second check; *Time 3*: the time elapsed from the time of the second check to the time when all the malicious jobs are killed. In our experiment, *Time 1* took most of the overall time. When the time window is large, the average time for *Time 1* becomes also large accordingly. However no matter how large is the window size, the sum of *Time 2* and *Time 3* taken for the agents to process accountability data and to communicate among them in the system was almost constant ranging from 2 to 4 seconds. These values are shown as the minimum time in Graph 6.9. When the attacks are launched right after the previous time window has just passed, *Time 1* takes as much as the time window shown as the maximum time in Graph 6.9.

The average times for each time window for the detection of randomly launched multiple attacks are shown as the medium time in Graph 6.9.

6.3.8. Detection and Protection from DDoS Attacks from the Source-End

This experiment deals with the attack model shown in Figure 5.2. The attack program used for this experiment is an apache HTTP server-benchmarking tool, '*ab*' [64]. This tool generates huge numbers of multiple page requests to an apache web server. The attacker's jobs were assigned at 50 and 80 CNs and executed *ab* to simultaneously send multiple page requests to a web server with the command $HOME/wlee/ab –kc 50 –t 900 http://wonjun.rcac.purdue.edu:8080/bigFile.

Each run at a CN performed 50 simultaneous multiple (with option *k* and *c*) requests from 50 CNs resulting in 2,500 (50×50) requests and from 80 CNs resulting in 4,000 (50×80) requests within one HTTP session. In order to increase the load, *bigFile* that is a big sized file was requested with the page. The attack duration ranged from 100 to 900 seconds (with option *t*), during which the requests by legitimate users were rejected. In our system, the time taken from the initiation to the termination of the attacks was 61 seconds for 50 CNs and 74 seconds for 80 CNs on average. However the time for 80 CNs did not necessarily take longer than the one for 50 CNs as shown in Graph 6.10. This time is measured as the interval from the time when the job starts to run to the time when the job completes due to the termination of all processes running on its behalf. During such interval, detection and protection were performed according to the following steps. First, the agent in the HN checked the PBS log file to find the CNs where the sub-jobs were assigned and sent the job information to the agents in these CNs. Second, the agent in each CN traced one by one the processes related to the PBS client process. This tracing was performed using the diagnostic and debugging tool *strace* [58]. Through *strace*, the agents collected the PBS job id and the name of the script submitted in this PBS job id and the name of the executable run in this script in turn. In the experiment, the last traced identifier of the process running as the executable was retrieved from the output file of trace. Third, the agent in each CN checked out the opened network files



Graph 6.10 Detection and Recovery Time for Different Attack Durations

using *lsof* [56]. This tool was executed every second to update information about files newly opened by the processes. The IP addresses of the destination for opened network files were recorded with the process ids in the log file and were searched by the identifier of the process running for the executable, that is, *ab* in the updated log file. The retrieved destination information was sent to the HN as a possible target IP address with the handle linked to the job. When deciding whether to send such information to the HN in a *light* alarm, a high threshold (i.e. entropy 0.95 for this experiment) was used because it is atypical to see many network files opened by a process running on behalf of the executable in a CN with the same destination resulting in very low entropy. The next step was performed by the agent in the HN. When the destination and handle information were sent to the HN from each CN, the agent in the HN calculated the entropy again and compared it with the threshold to issue the *moderate* alarm.

Finally the agent in each CN killed the currently running processes if the CN received a *critical* alarm from the agent in the entry node. Our experimental results reported in Graph 6.10 show that the attacked apache server was expected to be out of service for an interval ranging from 100 seconds to 15 minutes when the attacks were launched from the normal grid CNs without accountability agents. In the accountability grid computing system with 50 or 80 CNs, the attacks were stopped after 67 seconds on average. This Graph also shows that the detection and protection times are not dependent

on the attack duration. Therefore, even for long-lasting attacks, our system can detect the attack and take a response action within or around one minute.

6.3.9. False Positive Detection with Two Types of Threshold

In this experiment, we show how much the false positive rate detection typical of existing resource monitoring mechanisms can be reduced when integrated with our accountability system. We were interested in measuring the entropy of jobs over both the unique time window and unique handle and considered two types of thresholds, defined from two different types of entropy. The entropy, referred to as *H1* at *x*-axis in Graph 6.11, denotes the degree of randomness over the data obtained from the grid-node based strategy, while the entropy referred to as *H2* at *y*-axis is from the job-flow based strategy. When calculating *H1* the random variable *X* represents the time window at which the number of queued jobs is counted, while for *H2 X* represents the handle assigned to jobs. In Table 6.1, *H1* and *H2* are calculated from the data given in Table 5.2 in Chapter 5. Table 6.1-a shows the number of jobs associated with the unique time window at the end of each time window in a queue with the calculated entropy, respectively. Table 6.1-b shows the entropy of the queued jobs associated with the unique handle for an anomaly detected at the end of time window *t3*. Table 6.1-b can have a different number of jobs associated with the handle, resulting in many different values of entropy *H2* (for

Table 6.1 Data for *H1* and *H2* from Table 5.2 in Chapter 5. To apply Equation 6.1 – a. n is 33, n_0 is 3, *TW* Denotes Time-window, Q'd Denotes *queued* (Left Table); b. n is 24, and n_0 is 3 (Right Table).

TW	# of Q'd jobs	Entropy
t1	4	0.369
t2	5	0.412
t3	24	0.334
Sum of Entropy		1.115

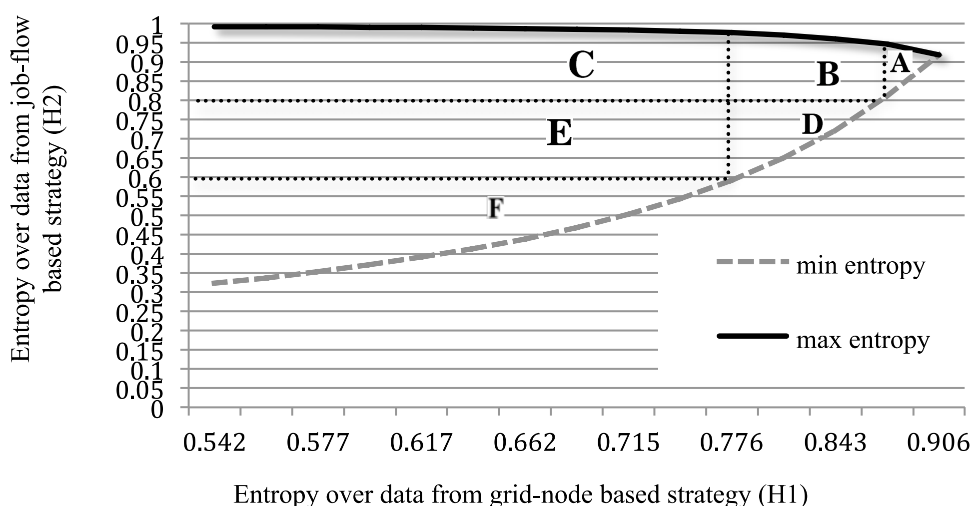
Handle	# of jobs	Entropy
ske	2	0.299
wai	2	0.299
abc	20	0.219
Sum of Entropy		0.817

$NE(H1) = 1.115/\log_2 3 = 0.703$, $NE(H2) = 0.817/\log_2 3 = 0.515$

example, *H2*=0.515 when *ske*=2, *wai*=2, *abc*=20; or *H2*=1 when *ske*=8, *wai*=8, *abc*=8, etc.) with respect to the anomalous data *H1*.

By employing another threshold (i.e., *th2*) defined in *H2*, the seemingly anomalous usage is classified as normal. In Graph 6.11, when the threshold (i.e., *th1*) defined in *H1* is equal to 0.906 (the highest value in the *x-axis*), an entropy value lower than *th1* indicates an abnormality regardless of what value entropy *H2* (i.e., *y-axis* value of any point in area A+B+C+D+E+F) has. However, if *th2* is equal to 0.8 and *th1* ranges between 0.86 (i.e., the *x-axi*s value that meets the *min entropy* line with the point 0.8 in the *y-axis*) and 0.906, we can expect that any case with *th1* and *th2* should be considered as normal because any point in area A is higher than 0.8 for such *th1*. If the *th2* equals to 0.8 and the *th1* is below 0.86, as much as area B+C out of B+C+D+E+F can be considered as normal because every point in area B+C has a value higher than 0.8 for such *th1*. Therefore we can expect a correction rate as much as area A+B+C out of A+B+C+D+E+F for all cases with *th2* that is equal to 0.8 and any value of *th1*. Likewise, if *th2* is equal to 0.6, we can expect that the false positive can be corrected as much as area A+B+C+D+E out of A+B+C+D+E+F.

This experiment thus shows that the accountability data collected by an agent according to the two different strategies can compensate the false positive problem



Graph 6.11 Probability Distribution for Normal Submissions with Two Different Types of Entropy

typical of the existing anomaly detection model.

6.4. An Environment of Accountability Data Queries

The purpose of providing an environment of accountability data queries is to analyze the accountability data and to visualize the moves of the malicious jobs. In this environment, the administrators are able to query different types of data from different types of tables that are created for accountability. In addition, when the agents detect DDoS attacks, the overall job-graph is formulated in visualization to provide better understanding of a job's movement from the submission node to execution nodes.

6.4.1. User Interface and Architecture

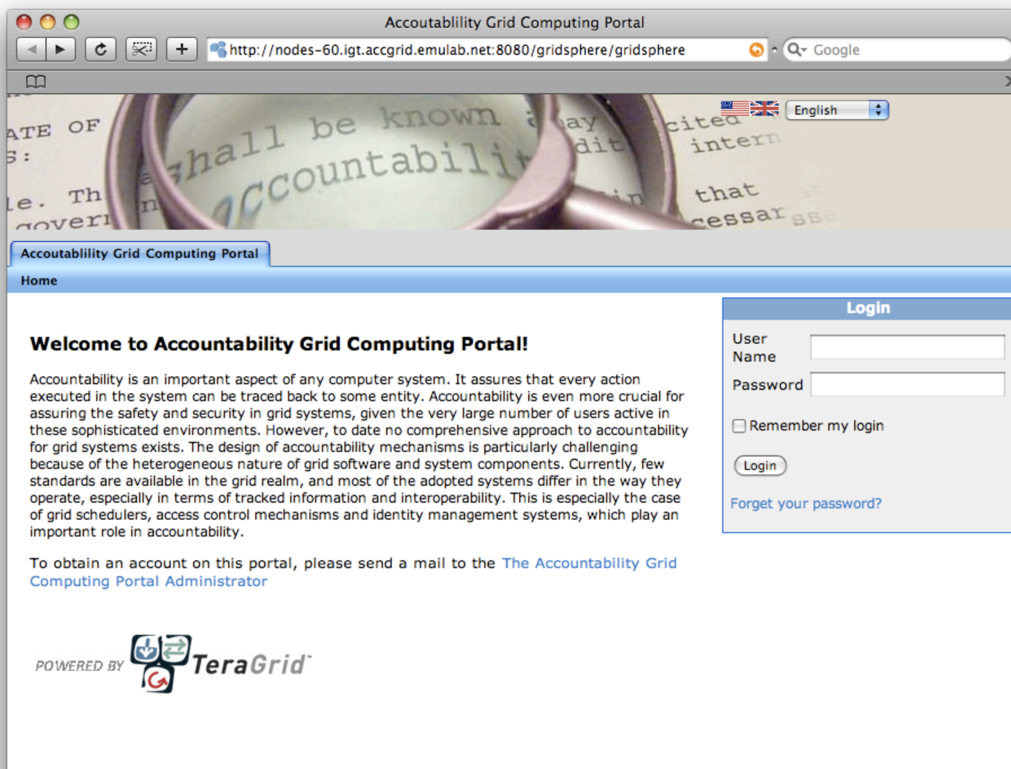
We used a gridsphere portal framework [49] that provides an open-source portlet based web portal. The querying environment is developed as a portlet web application and powered by apache tomcat. The gridsphere core portlets provide login, logout, and local settings, profile personalization, administration settings for creation of users, groups. Figure 6.7 captures screens of the first page for user authentication and the next page with major menu. Data-query portlet is written in Java and JavaServer Pages (JSP) technology. Data queries are sent to a selected database server running at each node.

The portal is available in each node. However, the root node of a job-graph can only show a complete job-graph in case that agents detect DDoS attacks because complete job-relation data are sent to the root node.

The interface of the accountability data query is composed of three tabs. They are 'Query Cover-record', 'Query Job-graph', and 'Query Record'.

Query Cover-record. The initial screen from this tab shows a cover-record that contains job-relation information such as where the job comes from, where the job goes to at the time of *timestamp* for *job-id* with *handle* information. Figure 6.8 shows a queried cover-record for a normal job at node, 60, which was chosen as a SP. Since node 60 is the entry node where the job is first submitted to, there is no *Job-Relation (FROM)* data. *Job-Relation (TO)* information shows that job (70e45498-954b-11e0-9f4d-001143e43a94) is

divided into (721cf0e0-954b-11e0-841f-96153c17c356) and (72acd4d0-954b-11e0-ace3-



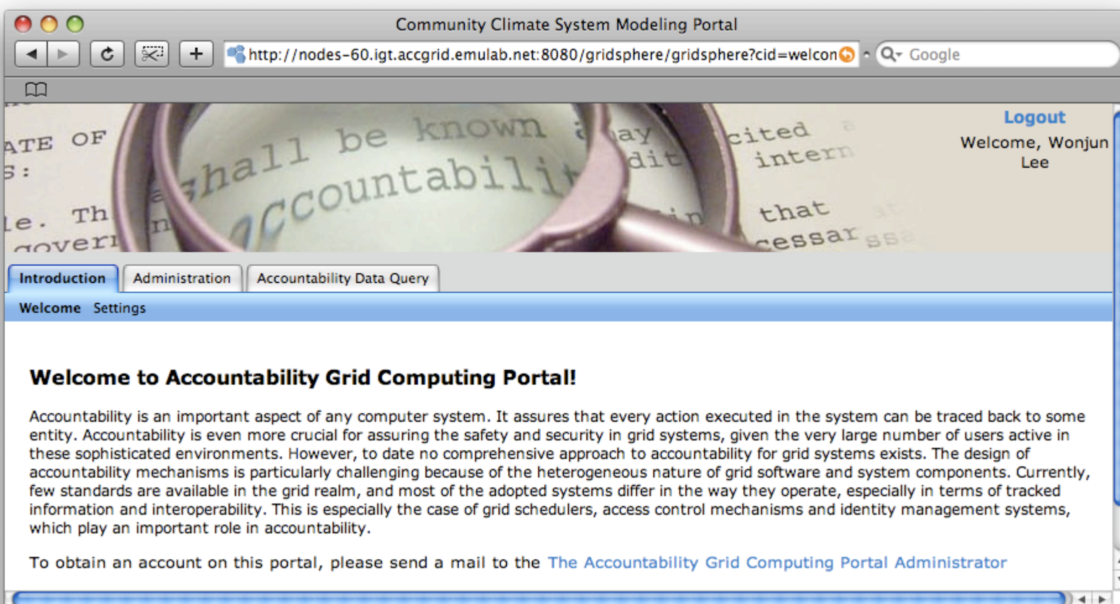


Figure 6.7 Main Screen of the Accountability Grid Computing Portal Before and After Login

Figure 6.8 A Cover-record for a Normal Job

c65637039387) and then moved to servers (155.98.39.5) and (155.98.39.6) respectively. *Job-Relation (LIST)* is only available when the misuse of resources is detected.

Query Job-graph. In this tab, a job-graph is drawn for each handle in case that cover-record contains *Job-Relation (LIST)*.

Query Record. In this option, the administrators can query accountability data in various ways. The accountability data collected at each node can be seen in a place from this interface.

6.4.2. Querying a Job-graph When Attacks are Detected

The job-graph is completed and visualized when the agents at each node send their job-relation data to upper nodes with alarms hierarchically up to the root node. The column *Job-Relation (LIST)* in Figure 6.9 shows pairs of job-id and its destination in order. In order to find out direct predecessor node, the agent in a node refers *Job-Relation (FROM)*. In entropy-based analysis, when the calculated entropy reaches below the threshold (i.e., when an alarm is issued), the agent sends *Job-Relation (TO)* data together with its job-id and server IP address to direct predecessor node. After receiving such job-relation data, the agent in upper internal nodes send the received *Job-Relation (LIST)* data

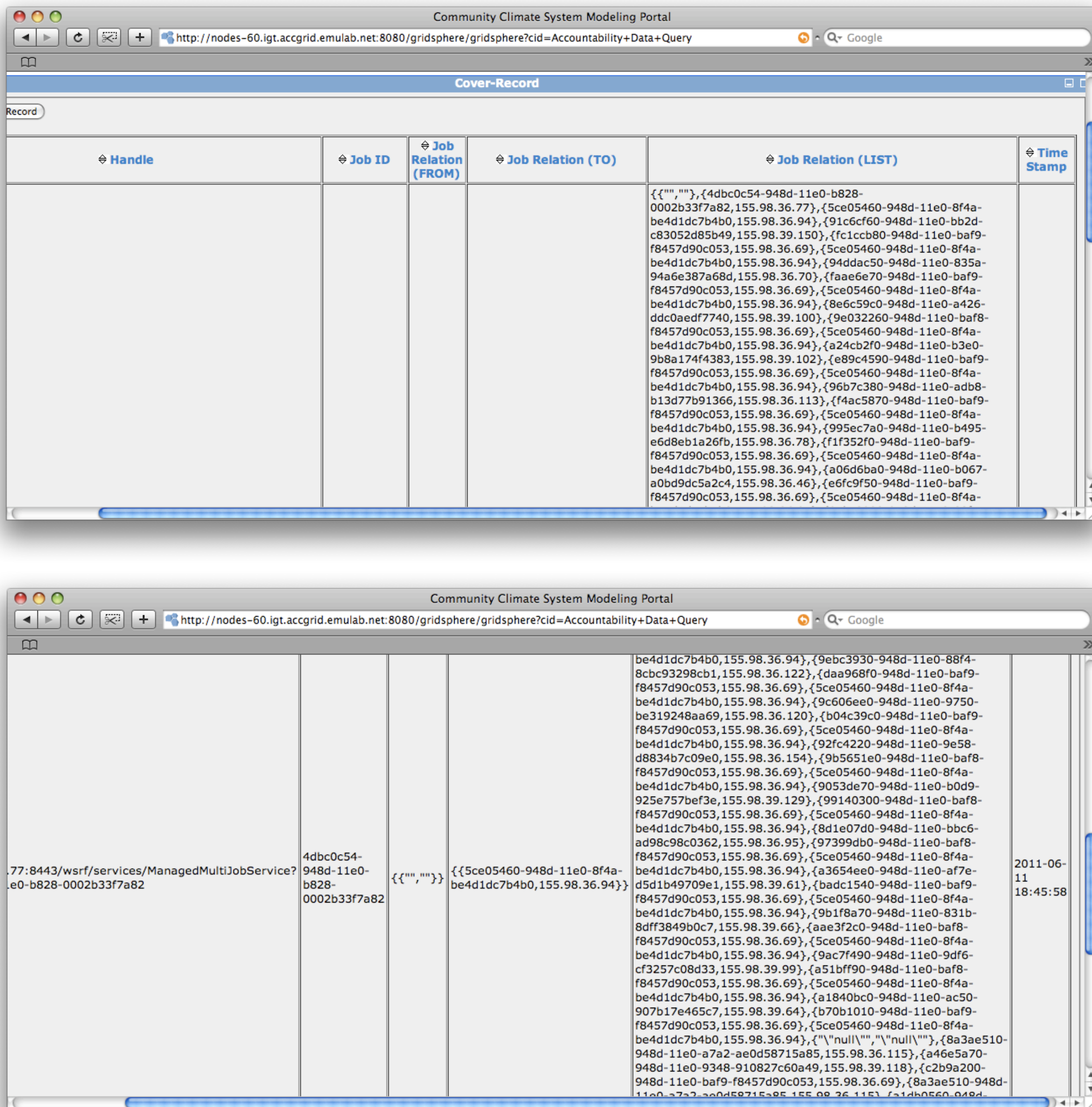


Figure 6.9 An Example of Job-Relation (LIST)

to its direct predecessor node by attaching its job id and IP address. This process is repeated at upper nodes until reaching a root node. Finally the agent in a root node collects all job-relation information.

Figure 6.10 is a grid topology constructed in Emulab test-bed. When an attacker exploits two clusters with two head nodes (nodes-20 and nodes-40) to attack a head node

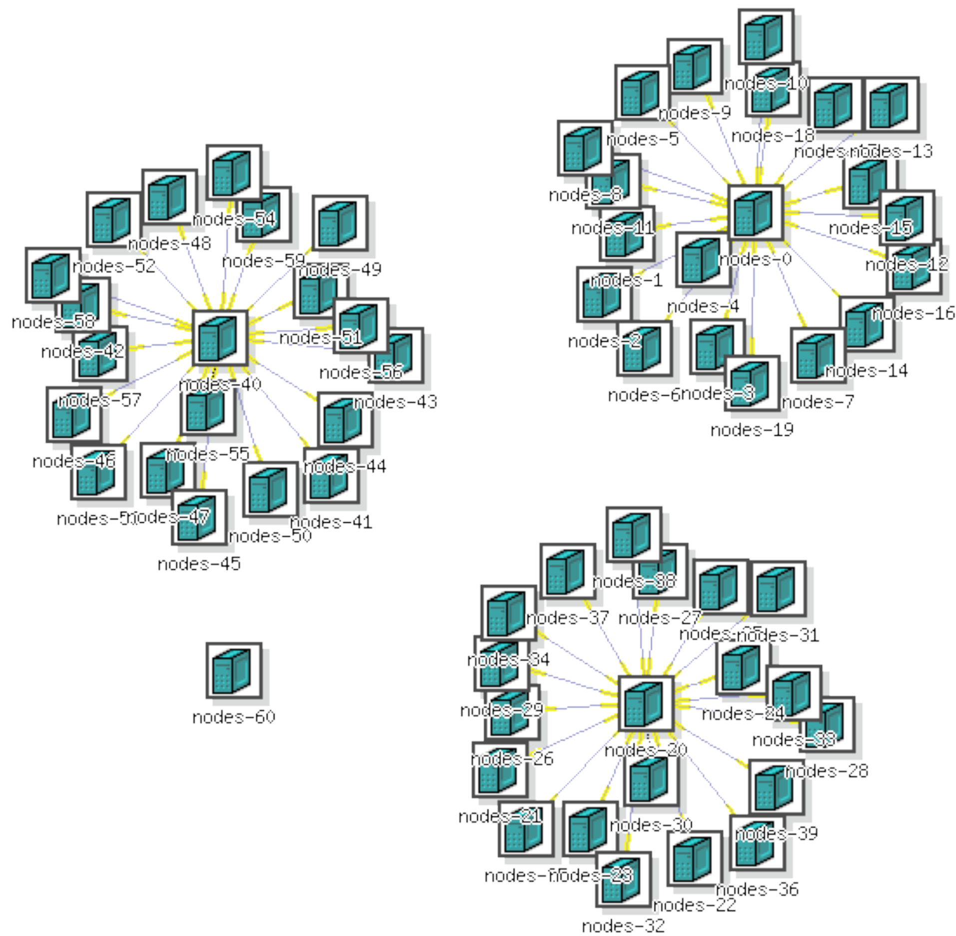


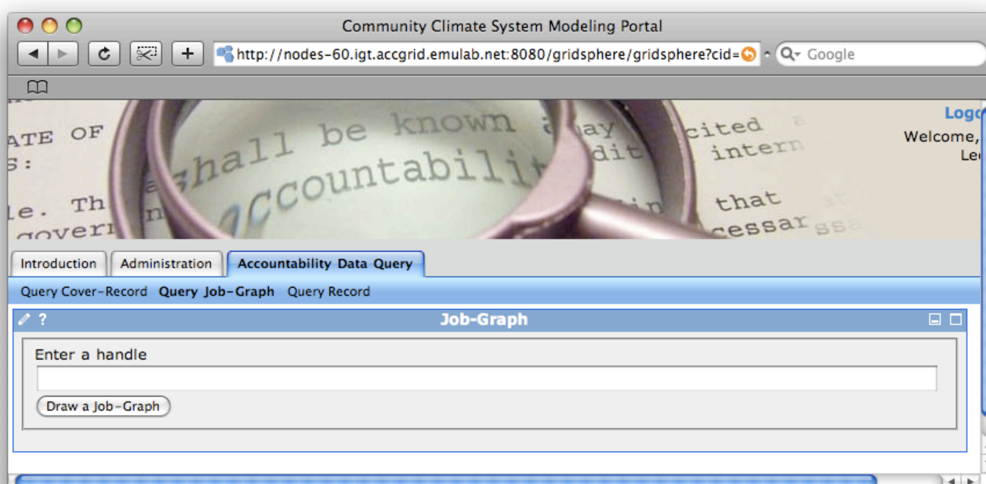
Figure 6.10 A Grid Topology from EmulabFigure 3.7 Cases of Comparisons with Shared Accountability

(nodes-0) in Figure 6.10, the constructed job-graph in the root node (nodes-60) reflects a part of constructed topology of Figure 6.10. This attack scenario is as follows. A job is submitted to a SP (nodes-60) and then assigned at a RP (nodes-20). The sub-job at nodes-20 is divided into 17 sub-jobs to be run at 17 nodes. One of 17 nodes is another RP (nodes-40) and a sub-job assigned at that RP is divided into 9 nodes again. The sub-job submitted to CNs from two RPs is rescheduled into a victim HN located in a RP (nodes-0).

When inserting a handle from Figure 6.9 at the portlet (Figure 6.11, top) and clicking the button, the portlet application draws a complete job-graph (Figure 6.11, down) by reading in the data in *Job-Relation (LIST)* from Figure 6.9. This job-graph shows job's movement from SP (nodes-60, 155.98.36.77) to the victim node (nodes-0, 155.98.36.69). Each node in the job-graph is represented with a subjob-id and its IP address.

6.4.3. Querying Accountability Data

An accountability agent in a node keeps its own database to collect accountability data. From this portlet, administrators can select any node (Step 1, Figure 6.12) to connect database in the node. In Step 2, available tables in the database are displayed. In our system, there are three types of tables. One table contains job-relation information shown in Figure 6.8. Other tables contain resource data and DDoS related information. The step 3 queries the column titles from the selected table so that the administrator can select the field of the table in multiple. The administrator can specify a condition in the same format as used in the postgreSQL query statement in step 4. An example such as querying *job-id* with *timestamp* from *accatable* for a specific *handle* (for a malicious job) in the victim node (nodes-0) is presented in Figure 6.13. In the result of the query (Figure 6.14), the sub-jobs used for attacks at each submitted time are displayed.



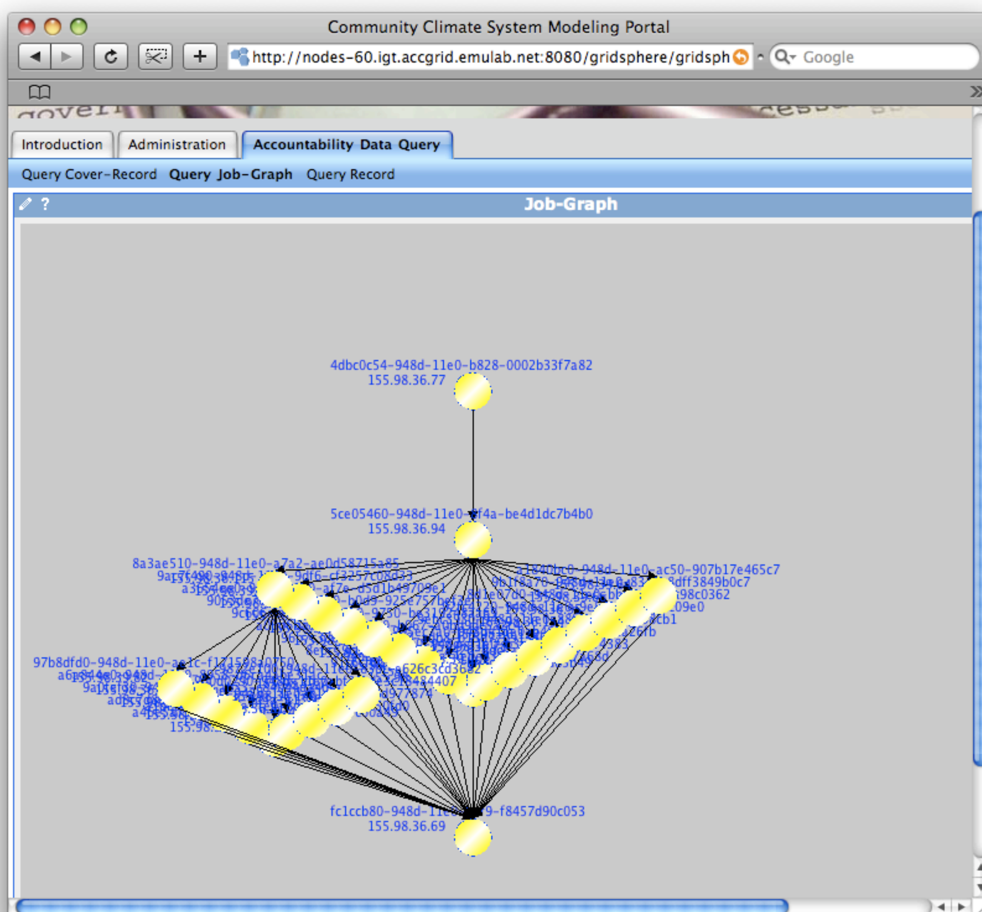


Figure 6.11 An Example of Job-graph

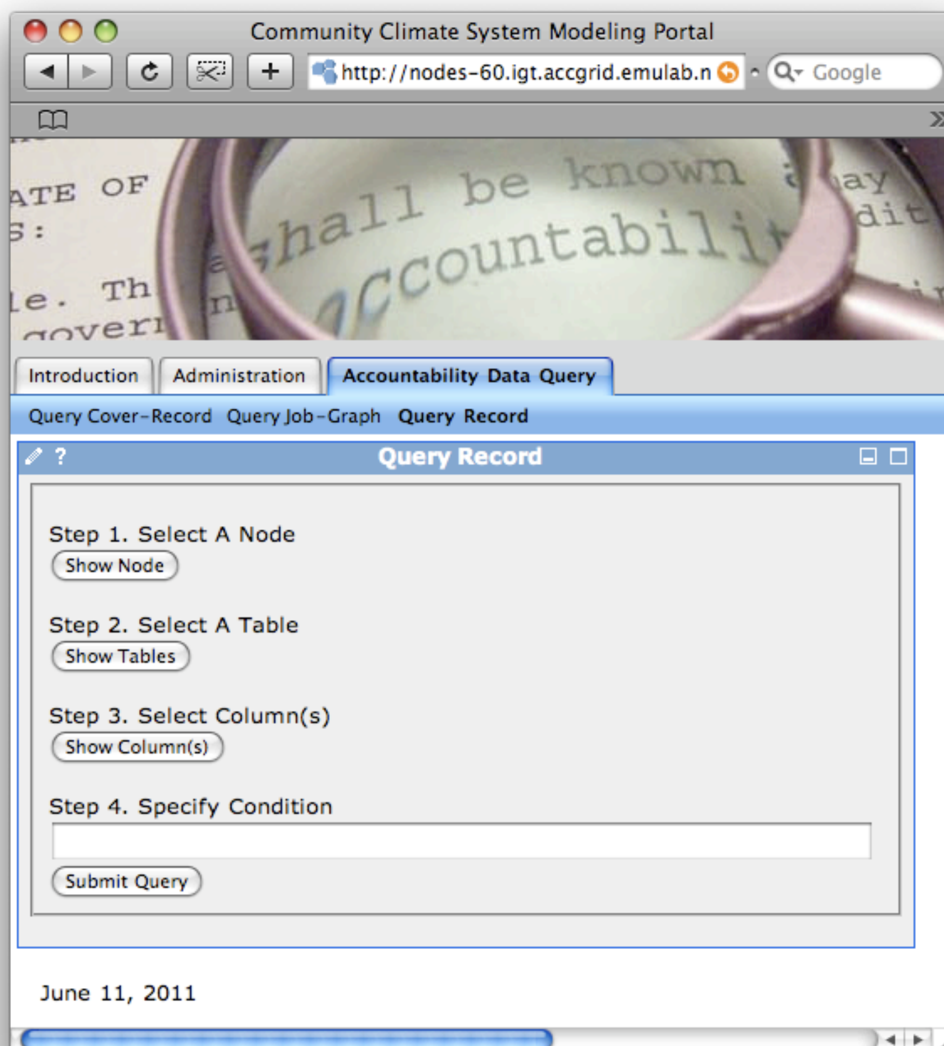


Figure 6.12 Initial Screen for Querying Accountability Data

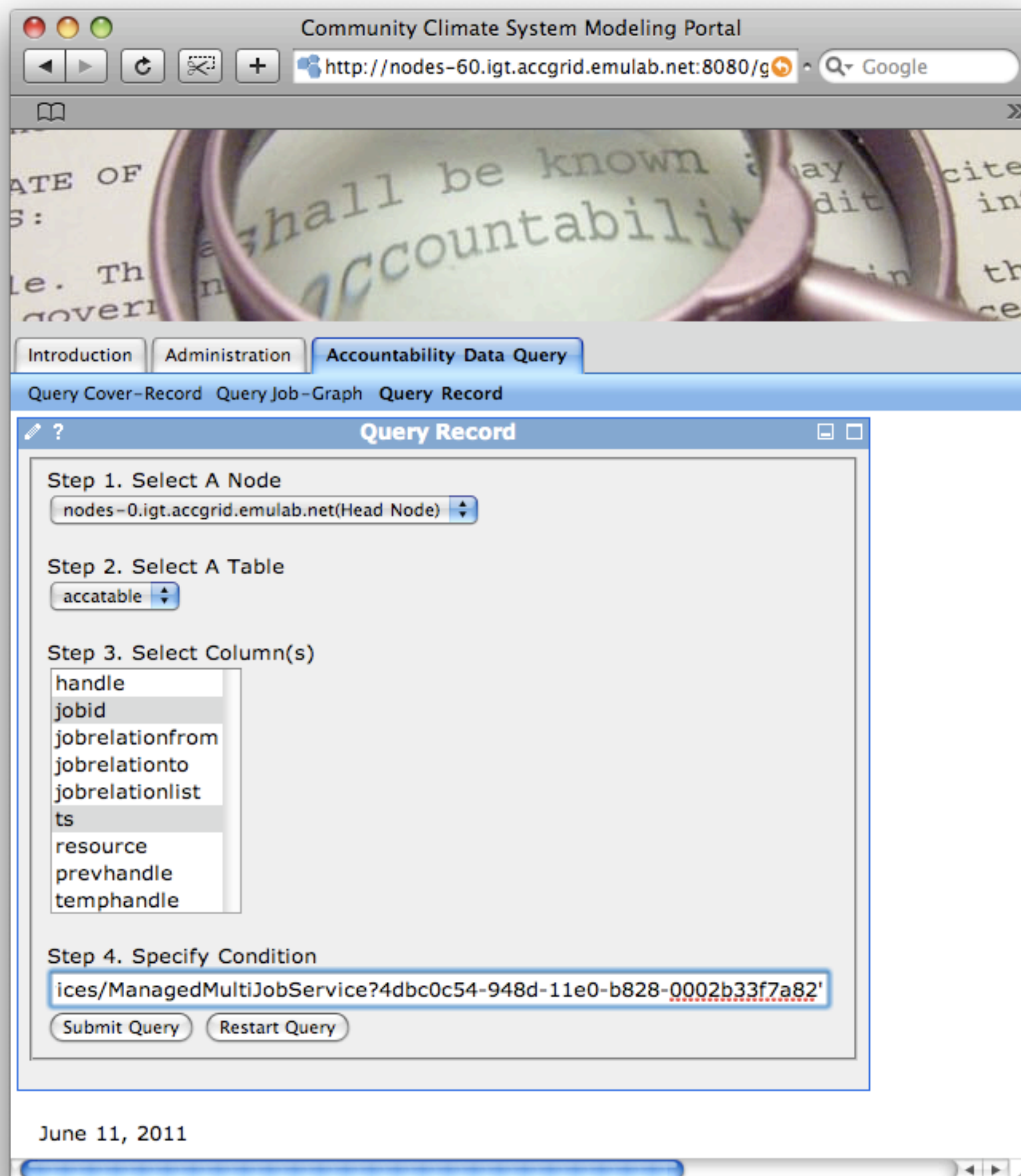


Figure 6.13 Selecting Options for Querying Accountability Records

Community Climate System Modeling Portal

http://nodes-60.igt.accgrid.emulab.net:8080/gr

Google

Introduction | Administration | **Accountability Data Query**

Query Cover-Record Query Job-Graph **Query Record**

Query Record

Step 1. Select A Node

nodes-0.igt.accgrid.emulab.net(Head Node)

Step 2. Select A Table

accatable

Step 3. Select Column(s)

ts

Step 4. Specify Condition

temphandle='https://155.98.36.77:8443/wsrf/services/ManagedMultiJobService?4dbc0c54-

Submit Query Restart Query

jobid	ts
99140300-948d-11e0-baf8-f8457d90c053	2011-06-11 18:47:59
97399db0-948d-11e0-baf8-f8457d90c053	2011-06-11 18:47:54
9b5651e0-948d-11e0-baf8-f8457d90c053	2011-06-11 18:48:04
9e032260-948d-11e0-baf8-f8457d90c053	2011-06-11 18:48:12
a51bff90-948d-11e0-baf8-f8457d90c053	2011-06-11 18:48:29
aae3f2c0-948d-11e0-baf8-f8457d90c053	2011-06-11 18:48:38
b04c39c0-948d-11e0-baf9-f8457d90c053	2011-06-11 18:48:48
b70b1010-948d-11e0-baf9-f8457d90c053	2011-06-11 18:48:54
badc1540-948d-11e0-baf9-f8457d90c053	2011-06-11 18:48:59

Figure 6.14 Result of the Query

7. RELATED WORK

Researchers have investigated accountability mostly as a provable property through cryptographic mechanisms. A representative work in this area is by [65]. Their approach, based on a logic language, proposes the usage of policies attached to the data and specified by the owner's data. The proposed logic differs from our approach in three main respects. First, their focus is on users' authorization data, while we deal with larger and richer types of accountability data. Our attention is on the nodes' site that needs to make sure jobs are properly submitted and not misused. Second, we do not impose any policy to be used at the submitter end, but let the agents collect the required information as needed. To this extent we employ a simple policy language to specify required data to collect. Third they do not report any actual implementation or experimental evaluation result, whereas we have a full working implementation and we have experimentally tested it.

Accountability has also been investigated in the context of electronic commerce protocols [66][67]. In particular Crispo and Ruffo propose an interesting approach related to accountability in the case of delegation. We do not directly consider delegation, although the graph shared mechanism implements a form of delegation process. Another interesting work is given by [68]. They propose layered architecture for achieving end-to-end trust and accountability. They adopt techniques for monitoring trust relationships over time so that abusive behavior can be tracked down. The authors drawn similar conclusions to ours, stating that current primitives for resource monitoring are not sufficient to support of fully accountability. However, they do not provide any specific language for specifying accountability policies, and they simply focus on users' data rather than providing aggregate accountability data combined with resource usage and job data.

Another interesting contribution is represented by the QUILL project [5]. A mechanism to capture provenance information during the execution of job in a distributed environment has been developed as part of such project. Although our work shares some commonalities with [5], we look at accountability as a general property instead of focusing on a specific technology, and ensure efficiently it in a distributed setting. Our solution does not rely on a specific underlying technology - we devise a general approach that can be mapped to actual mechanisms according to the specific technology considered. To that extent we introduce the notion of policies to support the specification of what to store and when, and provide a shared logging mechanism. We see the QUILL mechanism as a potential component of our system: it can be used to better extract data from Condor [14].

A number of techniques and tools have been proposed for monitoring grid resources, and services. However these systems restrict the notion of accountability to resource consumption monitoring or user account management. Currently many grid organizations typically adopt as resource monitoring tool one among the OGF-RU standard [69], Monalisa [70], Ganglia [6].

The OGF-RU standard represents an interesting approach. To share resources, sites exchange basic accounting and usage data in a common standard format defined by OGF. The record format facilitates the sharing of usage information for the purpose of job accounting among grid sites. Although our approach may seem similar to the approach by the OGF-RU standard, the major difference is that we focus on the connection among users, jobs and resources. Since our system uses two approaches to guarantee the principal's accountability, we achieve more fine-grained accountability than OGF-RU. Monalisa and Ganglia have very complicated and fault tolerant monitoring mechanisms. Agent plays similar roles as in our system. However such agents do not provide full accountability [68] because they do not provide information aggregated both horizontally (grid node based) and vertically (job-flow based). Moreover, they are not flexible in that they do not provide any policy language supporting the configuration of the accountability system while our accountability system is driven by accountability policies expressed in a policy language.

Resource monitoring alone is not enough to detect DDoS attacks and protect grid computing systems. Kar et al. [58] proposed an anomaly detection system for DDoS attacks in grids. This system uses an entropy-based model to detect anomalies caused by DDoS attacks and a grid topology model to implement the system. Compared to their approach, our approach covers a much broader set of grid layers and different types of DDoS attacks, while the approach by Kar et al. is limited to only a single network router. Though they employ additional thresholds defined from the entropy rate of the suspected flow in that router and the routers downstream, this mechanism only works when there are other objects to compare with. In addition, this router level detection cannot distinguish malicious job submissions from normal ones. Thus attacks introduced in Chapter 5 of this thesis will succeed, because the submitted jobs will be handled as legitimate jobs.

A related approach is by Xiang et al. [57], who proposed a distributed defense system composed of sub-systems to protect grids from DDoS attacks. Such system applies statistical methods to analyze the network characteristics. Like our system, when the system's sensors detect malicious activities, the detection system alerts the control system that then traces back the job through the system. Though this approach shares some similarities with ours, in our system each agent shares job-flow information with other agents as well as resource consumption information so that the two types of data can be combined to collect fine-grained accountability information. In addition, the system by Xiang et al. does not include any protection mechanism.

Chen et al. [71] propose an idea similar to the job flow discussed in this thesis. They propose a distributed approach to detecting DDoS attacks at the traffic-flow level. The job flow graph looks similar to their Change Aggregation Tree (CAT). However, the CAT differs from our job flow graph in several respects. First a CAT is constructed with the routers through which the attacks transit for detecting abrupt changes in traffic flows, while our job flow graph is constructed for accountability purposes with the nodes that a job traverses. Second, by analyzing the accountability data of jobs in the job flow graph, potential attacks can be prevented. Third, the centralized CAT servers play an important

role and make a decision while our job flow graph technique does not employ any centralized server and each node can make a final decision.

Another paper that discusses an idea similar to ours is by Mirkovic et al. [72]. Our detection mechanism in the source node is similar to their source-end detection (D-WARD) mechanism. However, when differentiating a malicious packet from a legal packet, they use semantic-based information such as 'one-way traffic', etc. while in our case, we use the behavior of the node in the context of the grid.

In terms of selecting different policies and resolving conflicts in distributed systems, Lupu et al. [73] propose an interesting approach. This approach aims at specifying implementable authorization policies, and then refining these policies into implementable actions, although policies are initially defined by the organization. Evolving a policy to the refined state seems similar to our work in terms of making the accountability policy close to a shared policy; however the final goals of this approach and ours are different. Lupu et al. focus on problems of conflict detection and resolution and propose various precedence relationships between policies to solve inconsistencies within the system. However our approach focuses on satisfying both the minimum level of accountability and the requested accountability for the shared policy when there is a conflict. In addition, their refining process is different from ours. In their approach the policy is refined from a high-level abstract level into an implementable policy, whereas in our approach the policy exists in an implementable form from the beginning and then evolves into an adapted policy after the minimum level accountability is guaranteed.

Another interesting approach for the resolution of policy conflict is by Davy et al. [74]. Their paper discusses how to facilitate conflict analysis of policies for services on multiple network devices. In this approach, ontology is generated from the information model of the system to embody knowledge about the relationships between policies. From such knowledge, policy analysis, incorporating policy selection and conflict analysis, is performed. Such approach uses application-specific information and knowledge required for conflict analysis. However the profile information in our approach is used not to detect conflicts but to select a level of accountability.

8. CONCLUSIONS

In this thesis, we introduced a distributed approach to achieve distributed accountability in grid computing systems. We introduced an architecture based on the notion of accountability agents that are software agents in charge of collecting a wide range of data and keeping track of connections among jobs, users, and resources. The accountability agents proposed in this work are distributed across the grid to collect accountability data and then coordinate to share the accountability data obtained locally based on a *shared policy*. The shared policy should be consistent among nodes to guarantee full accountability without conflicts. However each submitted job is exposed to a certain level of risk, according to the job type and importance. Similarly, according to the job's resource needs, nodes have different significance levels. In addition because of different node capabilities and a limited amounts of resources available for collection of the accountability data required by the shared policy, the shared policy should be different for each job and from node to node. To satisfy two properties of the shared policy, we have proposed a profile-based policy selection mechanism to adapt the shared policy to each node's ability within the requested and supported accountability while guaranteeing the minimum level of accountability.

Accountability data formed in distributed manner provide provenance information for real-time diagnostic of runtime anomalies. This real-time based diagnostic through data analysis plays an important role in helping to detect the source of malicious activities. To apply this obtained accountability data, we discussed different types of distributed denial of service attacks that could exploit grids and related detection strategies. Upon detection of an attack, the accountability agent system is able to protect the legitimate users' jobs by using accountability data.

We developed a fully working implementation of our accountability system and carried out extensive experimental evaluations. The experimental results show that our system does not impact the efficiency of current grid computing systems even for large-scale grids. In addition, our experiments showed that our system efficiently detects the attacks and is effective in protecting the normal jobs.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Webb, K., Hibler, M., Ricci, R., Clements, A., Lepreau, J. (2004): Implementing the emulab-planetlab portal: Experience and lessons learned. In WORLDS '04

[2] Foster, I., Kesselman, C. (1999): The Grid: *Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann: San Francisco, CA.

[3] Chivers, H. (2003): Grid Security: Problems and PotentialSolutions, Department of Computer Science, University of York

[4] Humphrey, M., Thompson, M. R. (2001): Security Implications of Typical Grid Computing Usage Scenarios, High Performance Distributed Computing

[5] Reilly, C. F., Naughton, J. F. (2006): Exploring provenance in a distributed job execution system. Proceedings of the International Provenance and Annotation Workshop, pages 237–245

[6] Massie, M. L., Chun, B. N., Culler, D. E. (2004): The Ganglia Distributed Monitoring System: Design, Implementation, and Experience, Parralel Computing, Vol. 30, Issue 7, Jul.

[7] Catlett, C. (2002): The philosophy of TeraGrid: building an open, extensible, distributed TeraScale facility. In Cluster Computing and the Grid 2nd IEEE/ACM International Symposium CCGRID

[8] Welch, V., Barton, T., Keahey, K., Siebenlist, F. (2005): Attributes, anonymity, and access: shibboleth and globus integration to facilitate grid collaboration. In: Proc of the 4th annual PKI R&D workshop

[9] Bader, D., Robert, P. (1996): *Cluster Computing: Applications*, Georgia Tech College of Computing. June.

[10] Foster, I., Kesselman, C., Tuecke, S. (2001): The Anatomy of the Grid, *Intl J. Supercomputing Applications*

[11] Foster, I., Kesselman, C., Tsudik, G., Tuecke, S. (1998): A Security Architecture for Computational Grids. *Proceedings of the 5th ACM Conference on Computer and Communications Security*, Nov., San Francisco, CA, USA

[12] Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W., Tuecke, S. (1998): A Resource Management Architecture for Metacomputing Systems. In *the 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 62—82

[13] Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnal, D., Tuecke, S. (2002): Data Management and Transfer in High Performance Computational Grid Environments, Parallel Computing Journal, Vol. 28 (5), May, pp. 749-771

[14] Frey, J., Tannenbaum, T., Foster, I., Livny, M., Tuecke, S. (2002): Condor-G: A Computation Management Agent for Multi-Institutional Grids, Cluster Computing, 5 (3). 237-246

[15] Karonis, N., Toonen, B., Foster, I. (2003): MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. Journal of Parallel and Distributed Computing

[16] Christie, M., Marru, S. (2007): The lead portal: a Teragrid gateway and application service architecture. Concurrency and Computation: Practice & Experience, 19:767-781

[17] Fortes, A., Figueiredo, J., Lundstrom, M. (2005): Virtual computing infrastructure for nanoelectronics simulation, Proceedings of the IEEE, 93:1839-1847, October

[18] Foster, I., Kesselman, C. (1997): Globus: A Metacomputing Infrastructure Toolkit, Intl J. Supercomputer Applications, 11(2):115-128

[19] Gropp, W., Lusk, E., Doss, N., Skjellum, A. (1996): A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing. 22(6):789-828

[20] Staples, G. (2006): TORQUE resource manager, Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, FL. Nov.

[21] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. (1994): PVM: Parallel Virtul Machine A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press

[22] Wallom, D., Spence, D., Tang, K., Viljoen, M., Jensen, J., Trefethen, A. (2007): ShibGrid, a Shibboleth based access method to the National grid service, AHM

[23] Morgan, R. L., Cantor, S., Hoehn, W., Klingenstein, K. (2004): Federated Security: The Shibboleth Approach. Educase Quarterly 27, 12–17

[24] Housley, R., Polk, W., Ford, W., Solo, D. (2002): Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 3280, Apr.

[25] Hallam-Baker, P. (2001): Security Assertions Markup Language. May, 14:1–24

[26] Khan, L., Awad, M., Thuraisingham, B. (2007): A New Intrusion Detection System using Support Vector Machines and Hierarchical Clustering, *The VLDB Journal* 16, 4, Oct., 507-521

[27] Barton, T., Basney, J., Freeman, T., Scavo, T., Siebenlist, F., Welch, V., Ananthakrishnan, R., Baker, B., Keahey, K. (2006): Identity Federation and Attribute-based Authorization through the Globus Toolkit, Shibboleth, Gridshib, and MyProxy, 5th Annual PKI R&D Workshop. Apr.

[28] Cantor, S. (2005): Shibboleth Architecture. http://shibboleth.internet2.edu/docs/internet2-mace-shibboleth-arch-protocols-latest.pdf

[29] Caelli, W., Longley, D., Shain, M. (1991): Information Security Handbook. London: Macmillan

[30] Bertino, E., Bettini, C., Ferrari, E., Samarati, P. (1998): An access control model supporting periodicity constraints and temporal reasoning. ACM Trans. Database Syst., 23(3):231-285

[31] Bertino, E., Bettini, C., Ferrari, E., Samarati, P. (1996): Supporting periodic authorizations and temporal reasoning in database access control. In Proc. International Conference on Very Large DataBases (VLDB), pages 472-483

[32] Shanmugasundaram, J. et al. (1999) Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, Edinburgh, Scotland

[33] Stevens, M., Sotrivo, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D. A., Weger, B. (2009): Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate, *CRYPTO, LNCS* 5677, pp. 55-69

[34] Laure, E. (2006): Programming the Grid with gLite, In Computational Methods in Science and Technology, Scientific Publisher OWN, pp 33-46

[35] Laure, E., Jones, B. (2008): Enabling Grids for e-Science: The EGEE Project, Grid Computing: Infrastructure, Service, and Application. CRC Press, Sep.

[36] Niinimaki, M., White, J., Cerff, W., Hahala, J. (2004): Using virtual organizations membership system with EDG's grid security and database acess, in *Procedings of the 15th International Workshop Database Expert System Application,* Sep., pp. 517–522

[37] Globus Security Advisory (2008): http://lists.globus.org/pipermail/security-announce/2008-April/000009.html

[38] SSH: unprivileged users may hijack forwarded X connections by listening on port 6010 (2008): http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=463011

[39] Grid Security vulnerability group (2011): http://www.gridpp.ac.uk/gsvg

[40] Groep, D., Koeroo, O., Venekamp, G. (2007): gLExec: Gluing Grid Computing to the Unix World. In *International Conference on Computing in High Energy and Nuclear Physics (CHEP),* volume 119 of Journal of Physics: Conference Series, Victoria, British Columbia, Canada, Sep.

[41] Kupsch, J. A., Miller, B. P., Heymann, E., Cesar, E. (2010): First Principles Vulnerability Assessment, *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, Chicago, IL, USA, Oct.

[42] Globus-job-manager vulnerability (2007): http://lists.globus.org/pipermail/security-announce/2007-May/000007.html

[43] Temporary file handling vulnerability (2006): http://lists.globus.org/pipermail/security-announce/2006-August/000003.html

[44] Altair Engineering PBS (2010): http://www.securityfocus.com/bid/41449/discuss

[45] Demchenko, Y., Gommans, L., Laat, C., Oudenaarde, B. (2005): Web-Services and Grid security Vulnerabilities and Threats Analysis and Model, *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, Nov., Seattle, WA, USA

[46] Belapurkar, A., Chakrabarti, A., Ponnapalli, H., Varadarajan, N., Padmanabhuni, S., Sundarrajan, S. (2009): Distributed Systems Security, *WILEY*

[47] Bloomberg, J. (2004): A Guide to Securing XML and Web-Services, ZapThink, LLC, Jan.

[48] Siddharth, S., Doshi, P. (2006): Five Common Web Application Vulnerabilities, SecurityFocus. http://www.securityfocus.com/infocus/1864

[49] Novotny, J., Russell, M., Wehrens, O. (2003): GridSphere: A portal framework for building collaborations. *In the first International Workshop o Middleware for Grid Computing*

[50] Vecchio, D. D., Hazlewood, V., Humphrey, M. (2006): Evaluating Grid Portal Security, *Super Computing (SC)* 2006, Nov., Tampa, FL, USA

[51] Chadwick, D. (2005): Authorization in grid computing, Information Security Technology Report, Jan.; 10(1): p. 33-34

[52] Karig, D., Lee, R. (2001): Remote Denial of Service Attacks and Countermeasures, Technical Report CE-L2001-002, Oct.

[53] Feitelson, D. G., Rudolph, L., Schwiegelshohn, U. (2005): Parallel Job Scheduling – A Status Report, Lecture Notes in Computer Science, Vol. 3277, Job Scheduling Strategies for Parallel Processing, Pages 1-16

[54] Gilbertson, K. (2002): Process Accounting, Linux Journal, vol 2002, issue 104, p.2

[55] Bandwidth Monitor (2010): http://www.bwmonitor.com

[56] Ward, B. (2004): How Linux Works, no starch press, p. 77-79, May

[57] Xiang, Y., Zhou, W. (2004): Protect Grids from DDoS Attacks, GCC 2004, LNCS 3251, pp. 309-316

[58] Kar, S., Sahoo, B. (2009): An Anomaly Detection System for DDoS Attack in Grid Computing, International Journal of Computer Applications in Engineering, Technology and Sciences, Vol. 1, Issue 2

[59] Freiling, F. C., Hoiz, T., Wicherski, G. (2005): Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks, In Proceedings of 10th European Symposium on Research in Computer Security, ESORICS, Milan, Italy, Sep.

[60] Huang, J., Kini, A., Reilly, C., Robinson, E., Shankar, S., Shrinivas, L., DeWitt, D., Naughton, J. (2006): An Overview of Quill++: A Passive Operational Data Logging System for Condor, Technical report, University of Wisconsin at Madison, Apr.

[61] Momjan, B. (2000): PostgreSQL: Introduction and Concepts. Pearson Education. Reading, MA

[62] Cover, T. M., Thomas, J. A. (2007): Elements of Information Theory, WILEY, second edition

[63] Steele (2008): http://www.rcac.purdue.edu/userinfo/resources/steele

[64] Apache software foundation (2010): Apache HTTP Server 2.2, Security and Server Programs, Fultus, Vol II, p. 130-133.

[65] Corin, R., Etalle, S., Hartog, J. I., Lenzini, G., Staicu, I. (2005): A logic for auditing accountability in decentralized systems. In 2nd IFIP TCI WG1.7 Workshop on Formal Aspects in Security and Trust (FAST), Toulouse, France, pages 187-201. Springer, August 22-27

[66] Crispo, B., Ruffo, G. (2001): Reasoning about accountability within delegation. In ICICS, pages 251-260

[67] Kailar, R. (1996): Accountability in electronic commerce protocols. IEEE Trans. Software Eng., 22(5):313-328

[68] Chun, B. N., Bavier, A. C. (2004): Decentralized trust management and accountability in federated systems. *37th Hawaii International Conference on System Sciences*, January

[69] Mach, R., Lepro-Metz, R., Jackson, S., McGinnis, L. (2006): Open Grid Forum (OGF) Resource Usage (RU) standard – Format Recommendation, Sep.

[70] Newman, H. B., Legrand, I. C., Galvez, P., Voicu, R., Cirstoiu, C. (2003): MonALISA: a distributed monitoring service architecture, Proceedings of the 2003 Computing in High Energy and Nuclear Physics

[71] Chen, Y., Hwang, K., Ku, W. (2007): Collaborative Detection of DDoS Attacks over Multiple Network Domains, IEEE Transactions on Parallel and Distributed Systems, Vol. 18, No. 12, Dec.

[72] Mirkovic, J., Reiher, P. (2005): D-WARD: A Source-End Defense against Flooding Denial-of-Service Attacks, IEEE Transactions on Dependable and Secure Computing, Vol. 2, Issue 3, Jul.

[73] Lupu, E. C., Sloman, M. (1999): Conflicts in policy-based distributed systems management, Software Engineering, IEEE Transactions on Vol. 25, issue 6, Nov.-Dec. Pages:852-869

[74] Davy, S., Jennings, B., Strassner, J. (2008): Using an Information Model and Associated Ontology for Selection of Policies for Conflict Analysis, IEEE international workshop on Policies for Distributed Systems and Networks, Palisades, NY

[75] Job Description Schema Doc (2004): http://www.globus.org/toolkit/docs/4.0/execution/wsgram/schemas/gram_job_description.html

APPENDIX

A. EXTENSIBLE MARKUP LANGUAGE (XML)

The XML grammar that is used at section 2.3.2 is introduced in appendix A.

XML declaration:

<?xml version=″1.0″ encoding=″UTF-8″?>

Comments:

<?--comments-->

Element:

<element>content</element>

Element Nesting:

<element_A><element_B>content</element_B></element_A>

Empty Element:

<info auther=”jame joyce” date=”1960-Jan-01” />

Attribute:

<element_name attribute_name=″attribute_value″>element contents</element_name>

Well-formed:

A well-formed document conforms to the XML syntax.

Valid:

A valid document, in addition, conforms to semantic rules either in an XML schema or user defined schema. If a document contains an undefined element, it is called not valid

Structured XML document:

Generic XML document contains a tree-based data structure.

Example:


```
<recipe name="bread" prep_time="10 mins" cook_time="2 hours">
  <title>Bread Recipe</title>
  <ingredient amount="8" unit="dL">Flour</ingredient>
  <ingredient amount="10" unit="grams">Yeast</ingredient>
  <ingredient amount="1" unit="teaspoon">Salt</ingredient>
  <instructions>
     <step>Mix all ingredients together</step>
     <step>Leave for five hour in warm room</step>
     <step>Turn on baking oven</step>
     <step>Leave oven for 30 minutes</step>
     <step>Bake in the oven at 450 for 30 minutes.</step>
  </instructions>
</recipe>
```

VITA

VITA

Wonjun Lee received a Master of Science degree in Electrical & Computer Engineering from Purdue University in December 2002. Before coming to Purdue, he worked for Samsung Electronics for two years at software development team of information & communication branch as a researcher. He started Ph.D. program at 2003 at Purdue, and began to work with Prof. Bertino from 2005 about the Ph.D. thesis topic. While pursuing the Ph.D. degree, he worked for Rosen Center for Advanced Computing (RCAC) from 2005 to 2010 as a research assistant. In RCAC, he participated in grid computing projects especially related to security issues such as identity management, authentication and authorization issues.