An Assessment of Accountability Policies for Large-Scale Distributed Computing Systems

Wonjun Lee
CERIAS, Purdue University
wlee@purdue.edu

Anna C. Squicciarini
The Pennsylvania State University
asquicciarini@ist.psu.edu

Elisa Bertino
CERIAS, Purdue University
bertino@purdue.edu

ABSTRACT
Grid computing systems offer resources to solve large-scale computational problems and are thus widely used in a large variety of domains, including computational sciences, energy management, and defense. Accountability in these application domains is an important requirement, in that it enables controlling activities of users and resource providers through the collection and analysis of accountability data. However, because it is not feasible to simply collect all the potentially useful data, we propose an accountability policy language to simplify administration tasks. In this paper, we show the efficiency of a system that we have developed to implement the language. We first present a short overview of the system, and then assess the proposed accountability policies based on the experiments.

Categories and Subject Descriptors
H.4.3 [Information Systems Applications]: Miscellaneous; D. 2.11 [Software Engineering]: Software Architectures – Languages, Patterns.

General Terms
Security

Keywords
Accountability, Policies, Grid, Distributed Systems, Scalability

1. INTRODUCTION
Grid computing systems allow researchers to use resources distributed across multiple domains. In such contexts, even when the users of a domain are fully trusted, the possibility of compromise and abuse always exists, because for example of vulnerabilities in system software and applications and configuration errors. Accountability is thus a crucial security feature to assure that every action performed in a system can be traced back to a real entity. Accountability becomes more essential in distributed systems where a large number of users actively interact, and more and more organizational units are dynamically involved by adding more resources. Detailed and comprehensive accountability data about users’ activities and actions provides a basis to analyze resource usage, and to find bottlenecks and detect security breaches [1]. Furthermore, through these data, system administrators can timely identify malicious users, damaged computer systems, and take proper protective actions [1]. However as the information gathered for accountability purposes can be potentially very voluminous, we need a tailored process to collect appropriate information according to administrator’s needs. The notion of accountability policy [4] addresses this requirement; such a policy specifies which information should be collected about jobs and users’ activities on the grid and when, and how this information is shared among grid nodes in different administrative domains.

In this work, we present our approach to policy-driven accountability. In order for accountability policies to be enforced, an accountability mechanism needs to be deployed. The accountability agent and the related database represent the most important components of our architecture. The accountability agent [4] is a software agent responsible for gathering accountability data and for sharing collected information with other agents, to keep an aggregate view of the job execution and the resource consumption. Every agent locally stores critical accountability information, which is also useful for postmortem analysis. We use a grid computing system as a reference environment for our accountability system. The key component of a grid computing system is a grid node. A grid node is any machine or cluster of machines that processes a job. For the purpose of our work, we define however a grid node as one component consisting of a computer system with appropriate software. Moreover, we assume, according to the common organization adopted by grid computing system, that each grid node has one of the following roles: Service Provider, Resource Provider, Head Node, and Computing Node. A Service Provider (SP) provides entry point where the job submission is initiated, such as portal or terminals. A Resource Provider (RP) provides available resources such as computational power, data, storage, and applications [5]. In our grid environment, we assumed that a cluster consists of one Head Node (HN) and multiple Computing Nodes (CNs). A HN schedules the submitted jobs based on the availability of CNs where the actual computation is performed [6].

A grid job is often composed of many sub-jobs that are executed in parallel in many CNs. The life cycle of a processed job can be very dynamic; millions of parallel jobs are split, and merged, and sometimes their execution loops unpredictably. Thus, in our system., we adopted a directed graph structure (job-graph) to represent the distributed activities of a job that have to be monitored for collecting accountability data.. Each vertex in a job-graph corresponds to the grid node where the job is submitted or assigned or processed. Directed edge denotes the job’s flow among nodes. The job-graph has a unique root node, denoting the starting point where the job is submitted and the analysis of the submitted job’s activities is performed.

In this extended abstract, we present our approach toward policy-driven accountability agents. We focus on the role of policies. Readers interested in our general architecture to accountability can refer to [4]. In the next section we provide a detailed description of our language for accountability policy. Section 3
presents our experiments and evaluations on the results followed by the conclusion.

2. ACCOUNTABILITY POLICIES

An accountability policy states actions that are relevant to accountability. These actions describe the main activities of agents. In this section, we present a logic formalism capturing the main components of our policy language. We begin with modeling the agents' basic actions using seven expressions. Actions can be of seven different types. The first four actions collect and send data required to build a view of job-graph local to the agent. The last three action specifications describe how to combine local data to obtain a complete view of the job-graph.

collect_job_data(x, state, data_set, storage): x denotes an agent; state denotes a set of job state; data_set denotes the set of data to be collected; storage denotes the data repository where the collected data have to be stored. This type of action specifies the information that the agent in the 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 already been submitted. However, the action should occur only when the specified state is reached. The state can be expressed at different granularity levels. A coarse grained expression may only consider the executing state of a 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; data_set denotes the set of data to be collected; time_constraints denotes a temporal expression; storage denotes the data repository where the collected data have to be stored. This 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 by Bertino et al. [2][3].

send_job_data(x, agent_job_relation, state, data_set, job_id): x denotes an agent; agent_job_relation denotes agents who will receive values of data_set; state denotes a set of job state; data_set denotes the set of data to be sent; job_id denotes job id. In order to build a partial view of the job-graph, agents at each node should send job relation information to the node to which the job flows.

receive_job_data(x, agent_job_relation, state, data_set, job_id): x denotes an agent; agent_job_relation denotes agents who will send values of data_set to x; state denotes a set of job state and: data_set denotes the set of data to be received; job_id denotes job id. In order to build a partial view of the job-graph, agents in each node should receive 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; agent_job_relation denotes agents to which request will be made by x; data_set denotes the set of data to be requested; job_id denotes job id. The agent of the root node or job-graph can trace every traversal of the job across the domains as if every grid node were in the domain local to the agent. To do this, the root node needs to request data to successor node's agents. Agents that 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; requester denotes agents who requested the values; data_set denotes the set of data to be forwarded; job_id denotes job id. The agents requested by the root node or the predecessor node are responsible to forward collected data to the requester. This action is executed only when the agent gets a request from the successor node.

combine_job_data(x, agent_job_relation, data_set, combined, job_id): x denotes an agent; agent_job_relation denotes agents who will forward values of data_set combined: data_set combined denotes the set of data to be combined; job_id denotes job id. Before forwarding data, the agent in the node combines collected data obtained from successor nodes with its data collected in local to eventually build a complete view of the job-graph by root node. When in general action expressions do not mandate any order of execution, some of them are meaningful only if executed in a certain sequence. For example, if one type of action expressions is of forward_job_data, then action expression should contain a combine_job_data stated as (forward_job_data⇒combine_job_data). Other examples are:

(send_job_data⇒collect_job_data);
(request_job_data⇒send_job_data).

The accountability policies are either local or shared. Local policies capture data that can only be obtained at local nodes, and do not require any form of coordination with other nodes across the system. Additionally, based on the above considerations, local policies can impose some temporal constraints. By contrast, shared policies specify which job information should be sent and requested upon state change of job and can include any combination of actions. Shared and local policies are specified according to the grammar. However we omit the syntax of grammar because of lack of space. In the following we provide two comprehensive examples about shared and local policies specified in the grammar.

Example 1. A job is submitted to Purdue University SP and then assigned for execution to RPs. A-state University, and B-state University. Purdue agrees to send job relation data (handle, job-id, subj_job-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) for every day during the week. The policies for such scenario are as follows:

Purdue shared_policy := send_job_data (agent@Purdue, agents_in_job_relationPurdue, active, data_setActive, job_id),
collect_job_data (agent@Purdue, active, data_setActive, DBPurdue),
agents_in_job_relationPurdue := agent@A-state (AND) agent@B-state, data_setActive := handle (AND) job-id (AND) subj-job-id (AND) RP-id (AND) timestamp (AND) local_policy := collect_resource_data (agent@A-state, data_setLocal, time_constraintsA, DBA-state), data_setLocal := memory consumption (AND) cpu time (AND) network bandwidth (AND) disk bandwidth, time_constraintsA := weekdays (AND) all_days

Example 2. (Continued from example 1) Suppose that a resource misuse (memory and cpu) is detected at A-state and reported to Purdue. Purdue then requests accountability information (handle,
The policies for such scenario are as follows:

\[ \text{Purdue} \] \text{shared policy}_{\text{Purdue}} := \text{request job data} (\text{agent@Purdue}, \text{agents in job relation}_{\text{Purdue}}, \text{dataSet}_{\text{Purdue}}, \text{job-id}), \text{dataSet}_{\text{Purdue}} := \text{handle} (\text{AND}) \text{job-id} (\text{AND}) \text{subjob-id} (\text{AND}) \text{RP-id} (\text{AND}) \text{time stamp} (\text{AND}) \text{memory consumption} (\text{AND}) \text{cpu time}, \text{combine job data} (\text{agent@Purdue}, \text{agents in job relation}_{\text{Purdue}}, \text{dataSet}_{\text{combined}}, \text{job-id}), \text{dataSet}_{\text{combined}} := \text{dataSet}_{\text{Purdue}} (\text{AND}) \text{dataSet}_{\text{A-state, Purdue}} (\text{AND}) \text{dataSet}_{\text{A-state}}, \text{dataSet}_{\text{Purdue}} := \text{dataSet}_{\text{A-state}}, \text{dataSet}_{\text{A-state}} := \text{dataSet}_{\text{B-state}}.

We implemented such policies as XML files; these files are created off-line by administrators and then stored in the local directory of the agents. In order to enforce policies, we embedded fine-grained monitoring primitives, encoded using Java, in a few routines of the grid middleware such as Globus Toolkits or GT4 [10]. To properly collect the data specified by the policies, the data must be gathered from different sources. Moreover, when enforcing a shared policy, each local agent must coordinate with other agents. Shared policies are evaluated whenever a job state change occurs. Precisely, at the SP/RP policies are evaluated when a notification from Globus Toolkits is received about a change in the job state. At other locations, the job state change always triggers a policy lookup process, to search for potential policies that need to be applied. Policy files are parsed into database tables only once and then these tables are queried each time the information they store is required. This approach saves on file accesses, and speeds up the process of identifying data that need to be collected. Such data is first locally stored, and then, specific agents’ functions are executed to send/receive the data as specified by the policies.

### 3. IMPLEMENTATION & EXPERIMENTAL EVALUATION

We implemented a prototype of the policy-based accountability system on the Emulab [7] test-bed. For this test-bed, we generated different grid topologies and connected the nodes using virtualized physical machines. Once the role of each node is decided, the required software is installed and configured. For example, in order to provide services/resources as a SP/RP, Globus Toolkit 4 is installed and configured; to play the head of a cluster, and as composing computing nodes, PBS scheduler is used and configured as a PBS server and workers. For lack of space, we do not provide additional details about the system’s architecture, and refer interested readers to [4]. We now discuss some experiments carried out as part of our evaluation.

One of the most important aspects to consider for any security technology is performance, because if the security technology interferes or degrades the existing system’s ordinary functions, it is not attractive to system administrators. Hence, one of our design goals is to minimally impact on the grid system’s ordinary activities. Therefore, our first experiment tested the accountability system’s scalability, by increasing the number of nodes. To measure how the accountability system performs as the system scales, we measured the job response time for both cases of running a job with an agent and without an agent at different clusters of different size. The first experimental result, reported in Figure 1, shows the response time and overhead of processing agent’s work. In this experiment, one intensive computing job is submitted to a cluster that is composed of 4 to 100 computing nodes to be run in parallel. It takes 285.67 seconds, and 288.5 seconds to finish the job at the cluster consisting of 4 compute nodes without and with agent respectively (leftmost bars). As the graph shows, the overhead of executing our system does not vary, regardless, of the size of the grid.

The good performance of our system is the result of an implementation strategy according to which most time-consuming functions work independently from GT4 or job scheduler. Asynchronous multi-threading implementation is in place. The experiments have confirmed that our accountability system is lightweight and does not interfere with the ordinary computation and activities of the system. Readers interested in other experiments for scalability can refer [11].

Our accountability system can generate a huge amount of fine-grained data, related to jobs, users and resources consumption. The next experiment (reported in Figure 2) shows how accountability policies can help in reducing such overhead, and by collecting only selected data. By using the policy, administrators can selectively choose data to save the storage and thus manage fewer amounts of data. However insufficient data may not guarantee full accountability. We show the relation between the policy complexity and data volumes collected at a certain node. For simplicity, we consider policy complexity in terms of the number of data elements to collect with the same action specification and the same attributes. The data volume of Figure 2 indicates the disk space taken for storing the obtained values of the policy elements at database. The gradient between the low and high complexity increases linearly when the number of submitted jobs increases linearly from 30 to 210. As shown, when a large number of jobs are submitted, the data volume drastically increases for complex policies while it does not change for small number of job submissions. When the system carries over
The next experiment (see Figure 3) analyzes the policy processing time for local and shared policies at a head node, where both local and shared policies are enforced. Policy processing includes reading the policy from the XML file, interpreting it, and collecting or searching elements specified at the policy. It does not include the time for database operations since we assume that updating the database for the same list of fields does not make a difference. Note that we used policies of the same complexity for both sample policies, even though they have a different structure with different number of elements. The average time for executing a local policy is approximately twice the time taken by the shared policy. This difference depends on the operations that have to be executed on log files. Such operations are required for the local policy in order to collect data. By contrast, collecting accountability data directly from the grid middleware, as it is the case for the shared policy, requires a much shorter time. As a result, we conclude that local policies are more expensive than shared policies. This result is confirmed by next experiment, which analyzes the search time required for the policy elements specified in different policies (see Figure 4). 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 Figure 3) 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. The last two experimental results are important in order to provide guidelines to administrators for the design of correct and efficient accountability policies, in that they indicate that local policies are to be carefully selected, due to their high overhead, and that policies collecting less resource and time consuming data items should be preferred.

4. CONCLUSIONS

We have developed a policy-based accountability system and analyzed the system performance with respect to policy evaluations. The experiments show that policies help optimizing resource utilization and are well suited for distributed systems. The results also show that local policies specified to collect resource usages for submitted jobs are more expensive than shared policies because of file operations. Based on this information, administrators may decide the accountability policies to adopt. However collecting extensive data for each job may reveal detailed information about user activities and resource usages. Protecting privacy of user's identities and their activities, while guaranteeing accountability, is part of our future work. Additionally, real time data analysis through provenance information is important part to give real time diagnostic of run time anomalies.

5. REFERENCES


