Investigating Global Behavior in Computing Grids

Kevin L. Mills and Christopher Dabrowski

National Institute of Standards and Technology
Gaithersburg, Maryland 20899 USA
{kmills, cdabrowski}@nist.gov

Abstract. We investigate effects of spoofing attacks on the scheduling and execution of basic application workflows in a moderately loaded grid computing system using a simulation model based on standard specifications. We conduct experiments to first subject this grid to spoofing attacks that reduce resource availability and increase relative load. A reasonable change in client behavior is then introduced to counter the attack, which unexpectedly causes global performance degradation. To understand the resulting global behavior, we adapt multidimensional analyses as a measurement approach for analysis of complex information systems. We use this approach to show that the surprising performance fall-off occurs because the change in client behavior causes a rearrangement of the global job execution schedule in which completion times inadvertently increase. Finally, we argue that viewing distributed resource allocation as a self-organizing process improves understanding of behavior in distributed systems such as computing grids.

1 Introduction

The Internet provides a communications infrastructure for distributed applications with global reach and massive scale. Already, designers have specified software components [1] that developers can use to construct and deploy computing and data grids [2], [3]. How will such distributed systems behave? One possibility is that distributed systems are complex adaptive systems [4] consisting of interconnected components, where change in any component propagates to many other components through feedback-driven interactions over space and time. Such interactions may arise through indirect coupling (e.g., sharing resources) exhibited as individual actors adapt their behavior based on information gained through feedback. Complex systems often exhibit the property of self-organization [4], which drives global system behavior (e.g., job scheduling and execution) from less organized states toward coherent patterns. We suspect self-organization will arise with increasing frequency as distributed systems pervade the globe. Unfortunately, little is known about how to detect, predict, and shape global behaviors in distributed systems.

Here, we study an important aspect of global behavior in grid systems, envisioned to offer high-performance computing as a commodity for those who require substantial processing cycles to design more effective drugs or engine components, to do financial risk analyses, to model global climate, to understand our universe, and so on. We consider distributed protocols for allocating processor resources deployed across a global grid. Our research shows such protocols can yield a self-organizing, global pattern of job scheduling and execution, as various independent clients sense the state of available processors and adapt accordingly. We investigate distributed resource allocation in a moderately loaded grid that is subjected to an attack intended to reduce substantially the available computing resources. We introduce a small change in client behavior to mitigate effects of the attack, but find unexpectedly that the global pattern of job execution degrades rather than improves. This result illustrates that surprising global behavior can arise in a distributed system, and motivates our interest in finding techniques to reveal, understand, and shape system behavior.

In this paper, we make three main contributions. First, we define a grid simulator combining model components representing selected, standard specifications. We chose specifications based on the current posture of the Global Grid Forum, which suggests that future grid systems will be built from a combination of web services [5] and open grid services [6]. For functions lacking completed specifications, we modeled components from the Globus Toolkit 4 [7], an available grid framework that provides a significant level of capability. Our simulator allows us to model a plausible distributed system in significant detail. Second, we show that a moderately sized grid can exhibit unanticipated and undesirable global behavior arising from adaptive processes. We illustrate that adaptation by many individual actors can lead to self-organization on a global scale. Third, we describe and apply a multidimensional analysis approach to reveal underlying causes for observed global behavior. The analysis approach is adapted from the physical sciences, where spatiotemporal analyses [8] have long been a standard technique to model system dynamics. While we use spatiotemporal analysis, we increase the number of dimensions to account for logical partitions within the system we study. For example, we consider completion times among various job classes over space and time (a 4D analysis). We find multidimensional analyses provide more insight into system behavior than can be obtained by summarization through averages and variances.

The remainder of the paper is organized as five sections. Section 2 outlines related work to simulate grid systems and to investigate distributed resource allocation in grids. Section 3 describes our analysis approach. We discuss our detailed grid simulation model in Section 4, before describing our experiment design and metrics in Section 5. Section 6 presents our simulation results, investigates causes underlying an unexpected outcome, and describes scheduling and execution of jobs in computing grids as a self-organizing process.

2 Related Work

While there is significant research on simulating grids, little of that work studies scalability, effects of failures and attacks, or global behaviors. SimGrid [9], GridSim [10], and MicroGrid [11] provide toolsets for simulating grid applications on large-scale networks. These grid simulations do not combine model components representing selected web services, Globus Toolkit 4 components, and open grid services. Further, these simulators aim mainly to provide overall assessment of performance in network protocols and middleware, rather than isolating causal behaviors that unexpectedly affect global performance.

Numerous researchers have investigated resource allocation in large (simulated) grids using a decentralized approach in which clients employ independent schedulers. For instance, Ernemann et al. [12] report results suggesting that geographically distributed grids improve overall response time. Various studies [12], [13], [14], [15] have applied market-based economic models to optimize resource use and minimize cost when scheduling jobs in distributed grids. Other researchers [16], [17], [18] have investigated prioritization schemes, considering factors such as quality-of-service and workflow requirements. Only a few grid-scheduling studies consider effects of uncertainty. Krothapalli and Deshmukh [19] consider performance of alternative scheduling approaches given partial information. Chen and Maheswaran [20] consider how resource failure affects scheduling. Subramani et al. [21] attempt to identify and address causes of unexpected performance degradations in grids. These studies rely extensively on summary measures of performance, and provide little insight into underlying global behavior. Our paper contributes to such investigations and demonstrates an analysis approach providing insight into global system behavior and causes.

3 Analysis Approach

We conduct simulations defined by a set of parameters (e.g., space, demand, negotiation strategy, and failure-response behavior) and observe system dynamics over time with respect to various logical partitions (e.g., event type and job class). We represent the entire system state as a multidimensional space. To investigate selected system dynamics, we project various views of this space, using a three-step procedure: (1) subset the space along dimensions of interest, (2) partition the subset into equivalence classes, and then (3) transform each equivalence class into measures of interest. Subsequently, we plot derived views in 2D, 3D, or 4D, depending upon the characteristics of the equivalence classes.

We represent system state(s) as a space, *U*, of multidimensional points, $\vec{x}$, i.e.,

$$U = \{(\vec{x} = (n_x, d_x, a_x, p_x, s_x, j_x, e_x, i_x, o_x))\}. \qquad \textbf{(1)}$$

Each of the dimensions is defined in Table 1. To explain our analysis procedure, we will derive two views used later in the paper to explore the effects of failure-response behavior (*s*) on two event types: reservations created (designated E_1) and task completions (designated E_2).

We begin by examining the effects of failure-response behavior on reservations created when demand (*d*) is 50% and the probability (*p*) of spoofing selected nodes is ½. We define a subspace, V_1, such that

$$V_1 = \{(\vec{x} \mid \vec{x} \in U \wedge d_x = 50 \wedge p = 0.5 \wedge e_x = E_1)\}. \qquad \textbf{(2)}$$

Next, we partition subspace V_1 into equivalence classes, Q_i, where every class consists of points with a common time interval (*i*), specifically

$$Q_i = \{\vec{x} \mid \vec{x} \in V_1 \wedge i_x = i\}. \tag{3}$$

Table 1. Definition of dimensions locating each point in system state space

Dimension	Variable	Range
Space	n	$1 \le n \le N$, where N is the number of observation points
Demand	d	$10 \le d \le 100$, where d mod 10 = 0
Negotiation Strategy	a	$a \in \{A_1,..., A_g\}$, where g = number of strategies
Spoofing Probability	p	$0 \le p \le 1$
Failure-Response Behavior	s	$s \in \{S_1,..., S_h\}$, where h = number of behaviors
Job Class	j	$j \in \{J_1,..., J_w\}$, where w = number of job classes
Event Type	e	$e \in \{E_1,..., E_z\}$, where z = number of event types
Time Interval	i	$1 \le i \le (T / I)$, where T is simulation run time and I is the observation interval size and $i = t/I$ for t = current simulation time
Observation	o	integer

Subsequently, we map portions of each equivalence class to a specific value by defining operators, $G_s(i)$, where

$$G_s(i) = \sum_{\substack{\vec{x} \in Q_i \\ s_x = s}} o_x. \tag{4}$$

This yields a 2D view, where one dimension represents a particular failure-response behavior (s) and the other dimension denotes specific time intervals (i) and each cell (s, i) contains an aggregate count of reservations created, obtained from equation 4. As discussed later, plotting such views for different s reveals large differences in the pattern of reservations created over time. To investigate such differences in more detail, we define a new view of the system state space to consider the evolution of task completion times for particular job classes (j).

We begin by defining the subspace, V_2, of interest:

$$V_2 = \{\vec{x} \mid \vec{x} \in U \wedge d_x = 50 \wedge p_x = 0.5 \wedge e_x = E_2 \wedge (s_x = S_1 \vee s_x = S_2) \wedge (j_x = J_1 \vee j_x = J_2)\} \tag{5}$$

and then a relation, R, to form equivalence classes on V_2:

$$\vec{x} R \vec{y} \Leftrightarrow (s_x = s_y \wedge j_x = j_y), \tag{6}$$

where $\vec{y} \in V_2$. Here, $V_2 / R = \{Q_k\}$ forms (k = 1, 2, 3, 4) equivalence classes, each combining one of two selected failure-response behaviors with one of two selected job classes. Next, we define a scaling factor, *f*, derived from the maximum observation in subspace V_2:

$$f = \max(o_x \mid \vec{x} \in V_2) + 1, \tag{7}$$

and then we define the following operators:

$$G_k(\vec{x}) = \underset{\vec{x}\in Q_k}{o_x} + 1 + (k-1) \times f. \tag{8}$$

4 Simulation Model

We find that simulation provides an excellent vehicle to investigate global behavior, for several reasons. First, simulation models allow construction of systems of large scale, which can be expensive to achieve in a test bed. Second, simulation models allow complete access to system state, which is impractical in a large deployed system. Third, simulation models provide rigorously controllable and repeatable conditions, which are difficult to ensure in a test bed of significant size. Fourth, simulation models allow incorporation of various levels of abstraction, while deployed systems typically include incidental complexity that is impractical to remove. Prior to conducting experiments, we verified our model for correct operation through extensive trials, removing several biases and errors in the process. In our experiment we used our model for qualitative analysis of system dynamics, rather then to make quantitative performance predictions; therefore, our verification process focused on ensuring correct interpretation of the standard specifications we modeled, rather than validating the model against measured performance data. In this section, we describe our simulation model, concentrating on the grid computing aspects.

4.1 Network and Web Services Model

We define a topology of sites, each located at a point (*x*, *y*, *z*). The *x*-*y* coordinates locate a site in the Internet and the *z* coordinate defines the distance in router hops from the site to the Internet. The model uses differences in *x*-*y* coordinates to compute Euclidean distances among sites, and then converts those distances to Internet router hops by assuming that routers are separated by a specified distance. The distance in hops between two sites is defined by the distance between the sites in Internet router hops plus the number of *z*-coordinate hops required for each site to reach the Internet. Messages flowing between sites are delayed in proportion to distance in hops. Messages flowing within sites incur simulated local-network transmission delays. Nodes are defined and allocated to sites. Each node has a *mailbox*, which simulates a sockets interface and related transport protocols, including multicast. Each node simulates CPU-execution time required by processes executing on the node. Nodes also include a standard set of services modeled after web services for messaging [22], addressing [23], and stateful resources [1]. At selected sites, our model deploys simulated information and index servers, which we model as service groups. We also define a two-level hierarchy of index servers, linked together through simulated query aggregators (modeled after the index service of Globus Toolkit 4) to form a monitoring and discovery service that grid clients use to discover the nature and location of available resources.

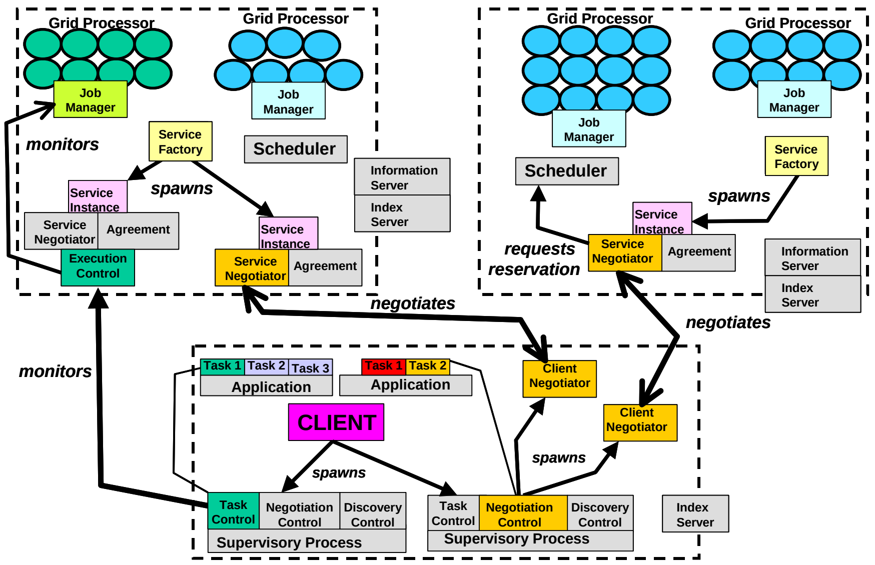

Fig. 1. Snapshot of system execution: client spawns supervisory processes for two applications

4.2 Grid Computing Model

At the application level, we model two main components, service providers and clients, found in well-known grid models (e.g., [2]). Service providers control availability of grid resources. Grid clients discover resources and enter into agreements for their use to execute client jobs. We describe these components and the procedures for reaching agreements and executing jobs. Figure 1 provides a simplified view of the model.

Service Providers. We designate selected sites as service-provider sites, where we deploy grid services, service factories, and schedulers. We model grid services in two parts: (1) application code, which executes jobs provided by clients; and (2) grid processors, which provide platforms upon which application code executes. Grid processors may be either clusters or vector computers, both capable of parallel execution. Each grid processor implements a simulated job manager (modeled after the Distributed Resource Management System [28]) that responds to job requests by locating and loading appropriate application code and obtaining input files from the requesting client. The job is then queued until its start time.

Each grid service has a *service description*, whose attributes include: type of application code (or task type) and descriptions of available grid processors on which the application code may run. Each *grid processor description* identifies processor type (*pType*: cluster or vector), available parallelism (or *pFactor*), and processor speed (*pSpeed* in cycles/second). A *service factory* manages each service description.

To advertise service availability, a service factory registers the service description with a local information server. Remote clients discover service descriptions through an indexing server and then contact an associated service factory to enter into agreements for services. For each client request, a service factory spawns a transient entity called a *service instance*, which provides a *service negotiator* to negotiate a service *agreement* locally on behalf of a remote client. The service instance contains an agreement document [29] and maintains negotiation status. If an agreement is reached, the service instance launches an *execution controller* to act as local proxy on behalf of the remote client. The execution controller submits jobs and status requests and reports job status to the client. We model each service instance (containing service negotiator, agreement, and execution controller) as a single container with lifetime determined by the negotiation duration or, if successful, by job length.

Each service-provider site contains a *scheduler* that controls reservation of CPU time on all grid processors within the site. For a client to obtain an agreement, a service negotiator must first reserve (through the site scheduler) time on an appropriate processor component. Each site scheduler is independent of other schedulers and accepts reservations on a first-come, first-serve basis with backfilling. All clients are given equal priority. As a policy, each scheduler attempts to allocate tasks with smaller *pFactors* to smaller clusters, thus saving large grid clusters for tasks requiring greater parallelism.

Grid Clients and Applications. We model each grid client as a set of independent applications, each with one or more tasks. While tasks in an application must execute sequentially in a workflow, each task contains subtasks that may execute in parallel. Each task is described by: task type (application code), *pFactor* (number of parallel processors required for subtasks), *pType* (cluster or vector), and *pCycles* (CPU cycles needed to run the task). The task duration, *tDuration*, may be computed as:

$$tDuration = tCycles \big/ (\, pFactor \times pSpeed \,) \qquad \textbf{(9)}$$

Clients create separate supervisory processes for each application. For each application task, the supervisor initiates service discovery, followed by negotiation to obtain an agreement to execute the task on a processor, and then monitors execution through a service instance. Since tasks are sequenced within an application, negotiation for a task is triggered when the previous task finishes. An application is complete when its last task finishes, at which time the supervisory process terminates.

We model supervisors as multiple components for: service discovery, agreement negotiation, and execution monitoring (including fault detection and recovery and job rescheduling). The discovery component activates on task completion to find the next task requiring services, and first queries a local index server for references to any remote information server with service descriptions matching task requirements. For each reference retrieved, the discovery component queries the associated information server to obtain matching service descriptions, which are then cached locally for use by the negotiation component. The negotiation component identifies the next incomplete task having no service agreement, ranks and selects cached discoveries for a task, and then creates a client negotiator for each selection. Selection criteria give higher priority to more recent discoveries and those that have not been tried previously. Similar to Condor-G [3], the negotiation component prioritizes discoveries during subsequent negotiations (as described below) to favor those that can execute the task sooner. If negotiation produces a successful agreement, a monitoring component registers with the service instance for notification of the outcome and accompanying output.

Agreement Negotiation. Negotiation commences when a client negotiator is created to obtain an agreement for a specific task. The negotiator, provided with a service description and the address of a service factory, queries the factory to obtain an agreement template, containing a set of possible agreement terms and, optionally, a list of creation constraints. We use creation constraints to convey existing reservations for grid processors associated with a service factory. Client negotiators use this information to determine an earliest possible start time. Execution ceases for any client negotiator obtaining an unacceptably late start time; otherwise, each client negotiator instantiates an agreement template for a selected task and end time on a subset of available processors, and then forwards the template (as an agreement offer) to a service factory, which spawns a service negotiator.

Our model allows negotiation to proceed according to one of two strategies: single-reservation request (SRR) or multiple-reservation request (MRR). In SRR, which closely follows WS Agreement [29], the service negotiator immediately acknowledges the client's request and contacts the scheduler to obtain a reservation, requesting start and end times on the processors specified in the agreement offer. If a reservation is granted, the service negotiator forwards an acceptance to the client; otherwise, the negotiator forwards a rejection (which can occur because multiple applications may be competing for the same processors). In SRR, an offer is considered obligating; thus, acceptance instantiates an agreement that both parties must observe. Independent of outcome, the negotiation terminates.

Negotiation Feedback. In MRR, negotiation may continue after initial rejection. MRR rejections contain an updated reservation list for applicable processors. A client negotiator may use this feedback to compose a "follow-up" offer to the service negotiator. This process may repeat, with additional follow-up offers, until an agreement is reached or negotiation is terminated. In contrast, SRR requires restarting negotiation to retrieve the template with the updated reservation list to use this feedback in preparing a new offer. The MRR strategy also differs from the SRR strategy in that client offers are not obligating. Using MRR, a client may create multiple client negotiators to simultaneously negotiate agreements for the same task and then accept the best one. The client may replace an existing observed agreement with a new agreement, if obtained prior to task execution. This again contrasts with the SRR strategy, where offers from client negotiators must be sequenced to prevent concurrent observed agreements. Each client was configured to negotiate with either MRR or SRR, while service negotiators handled both strategies. Both SRR and MRR adapt to feedback, as is characteristic of actors in self-organizing systems [4]. As we show in Sections 6.2 and 6.3, repeated attempts to secure services through negotiation lead to interactions where adaptation to feedback drives global resource allocation.

5 Experiment Description

We deployed our model in a simulated grid and conducted an experiment to compare the effectiveness and overhead of SRR and MRR. Below, we describe the experiment topologies, workload and design, and then define the metrics of comparison.

5.1 Experiment Topologies and Workload

In each experiment repetition, we generated a random topology by varying (uniformly between -4,000 and 4,000) the *x*-*y* coordinates ($z = 2$) of each site, which limited maximum inter-site distance to 16 hops. Each topology consisted of 42 sites: 30 service sites and 12 client sites. Each client site provided 25 applications for a total of 300 applications comprising 600 tasks. Each service site hosted a variable number of grid processors (allocation shown in Table 2) and one service factory to register service descriptions with a site-local information server. Service factories dynamically create service instances in response to client offers–one instance per client negotiator. Each service site contained one scheduler, and also contained a local information server and index server, which itself was subordinate to 12 index servers, one at each client site.

Table 2. Resources at Service Sites

Site Type	Processors at Site	Number of Sites
1	(1) 500-processor cluster	12
2	(2) 500-processor cluster	6
3	(1) 500-processor cluster (2) 1000-processor vector	6
4	(2) 500-processor cluster (1) 5000-processor cluster	6

Table 3. Description of Task Types

Task Type	*pType*	*pFactor*	*pCycles*
T1	Cluster	500	2.25e6
T2	Vector	1000	1.005e7
T3	Cluster	5000	2.50e7

Table 4. Description of Application Types

Application Type	Number, Type and Sequencing of Tasks
A1 and A2	(2) of task T1, executed sequentially
A3	(3) of task T1, executed sequentailly
A4	(1) of task T1 followed by (1) of task T3
A5	(1) of task T2

With no comprehensive studies of grid workflows to rely on, we chose relatively basic workflows in order to provide a simple baseline for analysis. We simulated applications consisting of one to three compute-intensive, parallelized tasks (each requiring between 500 and 5000 processors). We selected task types (Table 3) with execution times (average 1.38 hours/task) on the same order as observed in selected processor workload trace studies [30], [31]. Task definitions were combined to form application workflows of five types (Table 4).

We chose to experiment with a grid under moderate (50%) workload so that attacks would generate stress. Our attack model would not stress a lightly loaded grid, while a heavily loaded grid would already be operating under stress. We required each client site to have 25 applications (five instances of each application type), and calibrated simulated processor speeds to ensure these 300 applications consumed 50% of the capacity shown in Table 2. We activated service providers and applications after a random initial delay. Applications started after a further random delay (up to 2 hours) to simulate start of a workday. We assumed users requiring applications to complete within a 100,000 s deadline (just over one day), but allowed up to 200,000 s for applications to complete in order to measure the extent to which applications exceeded the deadline.

5.2 Experiment Design and Metrics

Initially, we considered effects on system performance of an attack carried out through *spoofing* by authorized but malicious service providers. Spoofing occurs when a bogus service factory at a miscreant site returns a faked template showing all the site's grid processors without reservations, leading a client negotiator to assume its task can be run immediately. This causes client negotiators to submit offers to the bogus service factory, which makes no further response, thus denying service to the client. Consequently, spoofed client negotiators time out (after 30 s) and terminate. Spoofing causes clients to lose time pursuing bogus resources, delaying application completion. We were interested to see how our simulated grid responded to this threat, and especially to discern performance differences between the two negotiation strategies. To assist clients, we defined an *adaptive failure-response* behavior in which clients react to negative feedback – service factories that caused timeouts were not retried for 5000 s, and after three consecutive timeouts a service factory was not retried for a specific task.

We configured our clients so that half negotiated under SRR and half under MRR. We subjected this configuration to three scenarios: (1) normal conditions (50% workload, no spoofing), (2) spoofing without failure response, and (c) spoofing with failure response. For each scenario, we executed repetitions of a simulated workday. Spoofing sites, chosen randomly with probability ½, remained so for the duration of a repetition. On average, 15 of 30 sites were spoofing in any repetition, eliminating half the system capacity and driving workload to 100%. Both spoofing scenarios were subjected to an identical sequence of 545 randomly generated topologies.

We measured two main aspects of system performance: application completion time (T_c) and overhead. We recorded frequency distributions (interval 10,000 s) for T_c across all topologies for each combination of scenario and negotiation strategy, and used those distributions to compute probability density functions (PDFs). For applications completing by goal T_g (= 100,000s), we computed average application duration, $\overline{D}_{app}$, as a proportion of T_g:

$$\overline{D}_{app} = \Big(\sum_{\substack{i=1 \\ [T_C \le T_g]}}^{N_{app}} (T_c - T_d)_i / T_g \Big) / N_{app} , \tag{10}$$

where T_d denotes time the discovery process commenced for application *i*, T_c is completion time for application *i,* and N_{app} is number of applications (with $T_c \le T_g$) in the experiment. We denote the average application duration across all repetitions as $\overline{\overline{D}}_{app}$. We also computed P($T_c \le T_g$), probability an application completes by T_g.

We measured overhead based on messages transmitted to complete negotiation. (Note that in these experiments we allowed up to five active client negotiators for each application task.) We defined as the minimum number of messages, $X_{ngt} = 25$, the expected minimum when five client negotiators activate simultaneously for a task using the SRR negotiation strategy. We computed the average O_{ngt} for a task as

$$\overline{O}_{ngt} = \sum_{i=1}^{N_{task}} \left[(M_{ngt})_i / X_{ngt} \right] / N_{task} , \tag{11}$$

where M_{ngt} is the number of negotiation messages counted to obtain agreements for task *i* and N_{task} is the number of tasks. $\overline{\overline{O}}_{ngt}$ denotes $\overline{O}_{ngt}$ averaged over all repetitions.

6 Results and Discussion

Table 5 summarizes system performance. Spoofing caused application duration to increase by 30% and $P(T_c \le T_g)$ to fall by 15%. Spoofing also caused negotiation overhead to increase fifteen times. However, incorporating adaptive failure response to combat spoofing decreased this overhead by about 50%, due to fewer interactions with spoofing sites. Unexpectedly, though, incorporating failure response caused an increase in application duration and a decrease in the probability of completing applications by T_g. This surprising result is supported by the probability density functions (PDFs) for application-completion times, plotted in Figure 2 for three scenarios: (a) no spoofing, (b) spoofing without failure response, and (c) spoofing with failure response. As expected, a large number of applications completed later when spoofing occurred, as client negotiators lost time making offers to bogus sites before eventually reaching legitimate sites. Unexpectedly, adaptive failure response led to a distinct right-shift in the PDF, as more applications completed later. This puzzling result required further investigation. To understand and explain the unexpected outcome, we identified topologies that exhibited the greatest performance degradation when using adaptive failure response. Then we selected one of those topologies for multidimensional analyses, using the approach outlined earlier in Section 3.

Table 5. Summary of system performance

	Application Duration $\overline{\overline{D}}_{app'}$			$P(T_c \le T_g)$			Negotiation Overhead $\overline{\overline{O}}_{ngt}$		
	Total	SRR	MRR	Total	SRR	MRR	Total	SRR	MRR
No Spoofing	0.355	0.363	0.348	1.000	1.000	1.000	2.02	1.15	2.89
Spoofing without failure response	0.660	0.728	0.526	0.845	0.728	0.963	30.74	40.09	22.05
Spoofing with failure response	0.712	0.816	0.612	0.800	0.711	0.890	17.46	18.90	16.11

6.1 Multidiminsional Analysis

Let failure-response behavior be $s \in \{NoFR, FR\}$, where *NoFR* signifies no failure response and *FR* signifies failure response, and let event types of interest be $e \in \{RC, TC\}$, where *RC* denotes reservations created and *TC* denotes tasks completed. We first examined effects of *s* on *RC*, instantiating subspace V_1 (equation 2) by choosing e_x=*RC*. We defined equivalence classes using equation 3, selecting observation interval *I*=1,000 s (the smallest granularity we could plot conveniently) over *T*=200,000 s to yield 200 equivalence classes $1 \le i \le 200$, summed across failure-response behavior *s* and job class *j*. Each equivalence class was then restricted to *s* = *FR* or *s* = *NoFR* and aggregated (equation 4) to generate two time series, as shown in Figure 3, revealing two different patterns of reservation creation. Most notably, Figure 3 shows, at *i* = 100 (*t* = 100,000 s), a large spike in reservations created appeared when failure response was employed. Figure 3 also shows that over the interval $1 \le i \le 50$ more reservations were created when failure response was used – while for the interval $51 \le i \le 100$, no reservations were created when failure response was used. These large shifts in the pattern of reservation creation suggested to us that the schedule of job executions was being altered.

To examine this at finer resolution, we defined additional subspaces to consider task completions (e_x=*TC*) under two negotiation strategies $a \in \{SRR, MRR\}$ for particular job classes $j \in \{A1T1, A1T2, A3T1, A3T2, A3T3\}$ chosen by selecting task types from Table 3 comprising application types A1 and A3 from Table 4. Using the approach shown in equation 5, we defined a subspace, V_2, to to consider the tasks for application A1. We then defined a relation, R_1, (similar to equation 6) to form eight equivalence classes $\{Q_k\}$, $k = 1 \ldots 8$, on V_2, with each partition representing a combination of failure-response behavior (*s*), job class (*j*), and negotiation strategy (*a*). Using equation 7, we determined a scaling factor, *f*, and operators as defined in equation 8 for each partition. Applying these operators yielded eight time series, shown in Figure 4 (a) and (b). Similarly, we defined another subspace, V_3, to which we applied R_1

to form 12 equivalence classes. We then determined a scaling factor and operators to yield 12 time series, shown in Figure 4 (c) and (d).

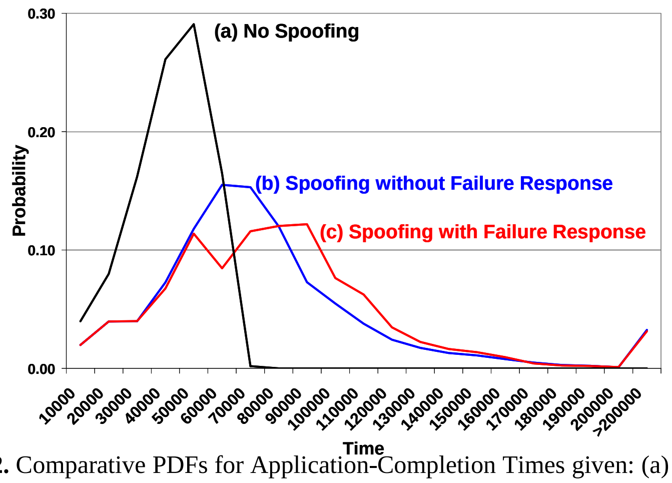

Fig. 2. Comparative PDFs for Application-Completion Times given: (a) No Spoofing, (b) Spoofing without Failure Response, and (c) Spoofing with Failure Response.

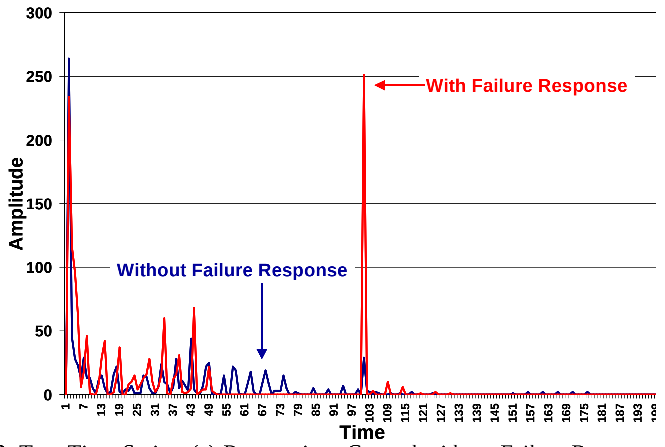

Fig. 3. Two Time Series: (a) Reservations Created without Failure Response and (b) Reservations Created with Failure Response

The 20 time series shown in Figure 4 reveal shifts in the temporal evolution of job completions. Specifically, when failure response was employed for clients using SRR, initial tasks for both applications (A1T1 and A3T1) scheduled and completed earlier than when failure response was not used, while their second tasks (A1T2 and A3T2) completed in roughly the same time range irrespective of failure-response behavior. Third tasks (A3T3) for clients using SRR were completed later when failure response was used. Employing adaptive failure response helped clients using SRR become more competitive in obtaining reservations earlier in time. As Figures 4(b) and 4(d) show, this led clients using MRR to have more difficulty obtaining early reservations for second (e.g., A1T2 and A3T2) and third (A3T3) tasks within an application. Thus, while the initial tasks for clients using MRR completed within the same time range regardless of whether adaptive failure response was used, second and third tasks were delayed when failure response was activated.

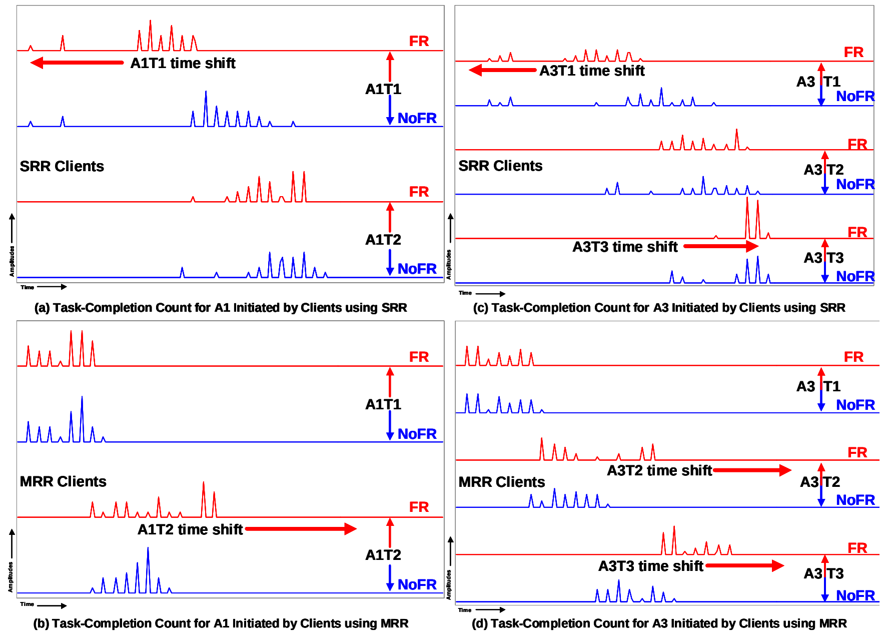

Fig. 4. Time Series of the Count of Task Completions Showing Time Shifts in the Execution Schedule When Adaptive Failure Response is used to Combat Spoofing Attack

6.2 Self-Organization and Adaptive Failure Response

This result demonstrates that taking a seemingly sensible action, here having clients resist spoofing, can lead to unanticipated system-wide performance degradation. In our example, introducing the adaptive failure response feedback mechanism (recall sec. 5.2) helped clients using SRR negotiation to become more competitive. This led to an unintentional reordering in the global schedule so that many second and third tasks executed later, and completion of jobs was delayed for most clients. This reordering arises from a self-organizing process in which numerous feedback-driven interactions among independent actors during negotiation form the global schedule. Multidimensional analysis allowed us to observe and explain this phenomenon.

We explored this behavior under several different conditions, as described in detail elsewhere [32]. First, we altered the scheduling algorithm to permit applications to reserve resources early (optimistically) for all tasks, rather than waiting until a previous task completed before seeking resources for the next task. Here, spoofing again caused performance degradation. Use of adaptive failure response caused further decline, with SRR clients benefiting at the expense of MRR clients. Multidimensional analysis of task completions revealed that when failure response was used, the global job execution schedule was again reordered so that second and third tasks executed later. Next, we introduced a different mix of applications and tasks, maintaining the basic workflow. Again, spoofing caused performance to degrade substantially, while use of failure response caused a slight degradation in which SRR clients benefited while MRR clients suffered. Multidimensional analysis of task completions revealed a similar task-reordering phenomenon to that observed with the original job mix. Finally, we defined a workload where every application consisted of only a single task, thus removing task dependencies in multi-task applications. While spoofing again increased application completion times, the introduction of failure response had little effect on the relative performance of MRR and SRR clients. Nevertheless, multidimensional analysis of task completions revealed underlying shifts in completion times, albeit in muted form. Thus, under the various conditions we examined, adaptive failure response either further degraded or failed to improve application-completion times, and a comparable reordering of job completions was observed in each case.

7 Conclusions

Scheduling and execution of jobs in a grid can exhibit a self-organizing behavior arising from distributed resource-allocation protocols. In the cases we studied, this self-organizing behavior leads to unexpected increase in application-completion times when adaptive failure response is used to combat a spoofing attack. Without tools such as multi-dimensional analysis to explore global behavior, much time and expense might be wasted attempting to identify and explain causes of this unexpected system performance. We believe that the key to understanding and controlling large, distributed systems is to view processes, such as distributed-resource allocation, as self-organizing. Doing so could provide a basis for creating measurement techniques and control algorithms to manage distributed systems of scale and complexity. Otherwise, with inadequate analytic tools, unanticipated consequences of underlying self-organizing processes will likely hamper adoption of promising technologies, such as grid computing.

In this paper, we explored a limited model of a computing grid that uses simple processes to allocate distributed resources among applications with basic workflows. Numerous researchers have proposed more complex regimes (such as market-based approaches) for distributed resource allocation, and we expect future proposed grid standards to include more nuanced algorithms, which may lead to other unexpected global behaviors caused by underlying self-organizing processes. Given this, we plan to explore behaviors that might arise if more sophisticated approaches are deployed. Subsequently, we will investigate techniques for influencing global behaviors in distributed systems.

References

1. The WS Resource Framework, V1.0. Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago (2004)
2. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid, An Open Grid Services Architecture for Distributed Systems Integration. Global Grid Forum (June 2002)
3. Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids. Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing. San Francisco (August 7-9, 2001) 55-67
4. Holbrook, M.B.: Adventures in Complexity: An Essay on Dynamic Open Complex Adaptive Systems, Butterfly Effects, Self-Organizing Order, Coevolution, the Ecological Perspective, Fitness Landscapes, Market Spaces, Emergent Beauty at the Edge of Chaos, and All That Jazz. Academy of Marketing Science Review (2003)
5. Web Services Architecture. W3C Working Group Note (February 11, 2004)
6. The Open Grid Services Architecture, Version 1.5. Global Grid Forum (March 10, 2006)
7. I. Foster et al.: A Globus Primer or, Everything You Wanted To Know About Globus But Were Afraid to Ask, an Early and Incomplete Draft (May 8, 2005)
8. Bak, P.: How Nature Works: the science of self-organized criticality. Copernicus, New York (1996)
9. Legrand, A., Marchal, L., Casanova, H.: Scheduling Distributed Applications: The SimGrid Simulation Framework. Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03). Tokyo (May 12-15, 2003) 138-145
10. Buyya, R. Murshed, M.: GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid Computing. Concurrency and Computation: Practice and Experience. Vol. 14. (2002) 1175-1220
11. Liu, X., Xia, H., Chien, A.: Validating and Scaling the MicroGrid: A Scientific Instrument for Grid Dynamics. Journal of Grid Computing. Vol. 2, No. 2 (2004) 141–161
12. Ernemann, C., Hamscher, V., Yahyapour, R.: Benefits of Global Grid Computing for Job Scheduling. Proceedings of the Fifth IEEE International Workshop on Grid Computing (GRID 2004). Pittsburgh. (November 8, 2004) 374-379.
13. Wolski, R., Brevik, J., Plank, J., Bryan, T.: Grid Resource Allocation and Control Using Computational Economies. In Berman, F, Fox, G., Hey, T. (eds.): Grid Computing: Making the Global Infrastructure a Reality. Wiley and Sons, New York. (2003) 747–772
14. Gomoluch, J., Schroeder, M.: Market-based Resource Allocation for Grid Computing: A Model and Simulation. Proceedings of the First International Workshop on Middleware for Grid Computing. Rio de Janeiro. (June 16-20, 2003) 211-218
15. Yeo, C.S., Buyya, R.: Service Level Agreement based Allocation of Cluster Resources: Handling Penalty to Enhance Utility. Proceedings of the 7th IEEE International Conference on Cluster Computing. Boston. (September 27-30, 2005)
16. In, J., Avery, P., Cavanaugh, R., Ranka, S.: Policy Based Scheduling for Simple Quality of Service in Grid Computing. Proceedings of the Eighteenth International Parallel and Distributed Processing Symposium (IPDPS'04). Santa Fe. (April 26-30, 2004) 23
17. He, X., Sun, X., Von Laszewski, G.: A QoS Guided Scheduling Algorithm for Grid Computting. Journal of Computer Science and Technology, Special Issue on Grid Computing. Vol. 18, No. 4 (2003) 442-450
18. Cooper, K., et al.: New Grid Scheduling and Rescheduling Methods in the GrADS Project. Proceedings of the Eighteenth International Parallel and Distributed Processing Symposium (IPDPS'04). Santa Fe. (April 26-30, 2004) 199
19. Krothapalli, N. Deshmukh, A.: Dynamic allocation of communicating tasks in computational grids. IIE Transactions. Vol. 36, No. 11. (2004) 1037-1053

20. Chen, H. Maheswaran, M.: Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems. Proceedings of the Sixteenth International Parallel and Distributed Processing Symposium (IPDPS 2002). Fort Lauderdale (April 15-19, 2002)
21. Subramani, V., Kettimuthu, R., Srinivasan, S., Sadayappan, P.: Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC-11 '02). Edinburgh (July 24-26, 2002) 359
22. SOAP V1.2 Part 1: Messaging Framework. W3C Recommendation (June 24, 2003)
23. WS Addressing. BEA Systems Inc., International Business Machines Corporation, and Microsoft Corporation, Inc. (March, 2004)
24. WS Resource Lifetime, V1.1. Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago (March, 2004)
25. Publish-Subscribe Notification for Web Services, V1.0. Akamai Technologies, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation, SAP AG, Sonic Software Corporation, Tibco Software Inc. and The University of Chicago (March 2004)
26. WS Services Topics, V1.0. Akamai Technologies, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation, SAP AG, Sonic Software Corporation, Tibco Software Inc. and The University of Chicago (March, 2004)
27. WS Service Group, V1.0. Computer Associates International Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago (March, 2004)
28. Distributed Resource Management Application API Specification 1.0. Global Grid Forum (June, 2004)
29. Web Services Agreement Specification (WS-Agreement). Global Grid Forum (September, 2005)
30. Parallel Workloads Archive. The Hebrew University of Jerusalem. http://www.cs.huji.ac.il/labs/parallel/workload/
31. Shan, H., Oliker, L.: Job Superscheduler Architecture and Performance in Computational Grid Environments. Proceedings of the 2003 ACM/IEEE Conference on Supercomputing. Phoenix (November 15-21, 2003) 44
32. Mills, K., Dabrowski, C.: Investigating Global Behavior in Computing Grids: the Extended Report. Draft technical report. U.S. National Institute of Standards and Technology (Available from the authors).