Chapter 1

PERFORMANCE INFORMATION SERVICES FOR COMPUTATIONAL GRIDS*

Rich Wolski

Lawrence J. Miller

Graziano Obertelli

Martin Swany
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

Grid schedulers or resource allocators (whether they be human or automatic scheduling programs) must choose the right combination of resources from the available resource pool while the performance and availability characteristics of the individual resources within the pool change from moment to moment. Moreover, the scheduling decision for each application component must be made before the component is executed making scheduling a predictive activity. A Grid scheduler, therefore, must be able to predict what the deliverable resource performance will be for the time period in which a particular application component will eventually use the resource.

In this chapter, we describe techniques for dynamically characterizing resources according to their predicted performance response to enable Grid scheduling and resource allocation. These techniques rely on three fundamental capabilities: extensible and non-intrusive performance monitoring, fast prediction models, and a flexible and high-

*This work was supported, in part, by a grant from the National Science Foundation's NGS program (EIA-9975020) and NMI program (ANI-0123911) and by the NASA IPG project.

performance reporting interface. We discuss these challenges in the context of the Network Weather Service (NWS) – an on-line performance monitoring and forecasting service developed for Grid environments. The NWS uses adaptive monitoring techniques to control intrusiveness, and non-parametric forecasting methods that are lightweight enough to generate forecasts in real-time. In addition, the service infrastructure used by the NWS is portable among all currently available Grid resources and is compatible with extant Grid middleware such as Globus, Legion, and Condor.

1. Introduction

The problem of scheduling and resource allocation is central to Grid performance. Applications are typically composed of concurrently executing and communicating components resulting in the "traditional" tension between the performance benefits of parallelism and the communication overhead it introduces. At the same time, Grid resources (the computers, networks, and storage systems that make up a Grid) differ widely in the performance they can deliver to any given application, and this deliverable performance fluctuates dynamically due to contention, resource failure, etc. Thus an application scheduler or resource allocator (whether it be a human or an automatic scheduling program) must choose the right combination of resources from the available resource pool while the performance and availability characteristics of the individual resources within the pool change from moment to moment.

To assign application components to resources so that application performance is maximized requires some form of resource valuation or characterization. A scheduler must be able to determine the relative "worth" of one resource versus another to the application and choose the ones that are most valuable in terms of the performance they deliver. If the scheduling decision is to be made by an automatic scheduling or resource allocation program (e.g. AppLeS (Berman et al., 1996; Spring and Wolski., 1998) or GrADSoft (Berman et al., 2001; Petitet et al., 2001; Ripeanu et al., 2001)) this valuation must be in terms of quantifiable metrics that can be composed into a measure of application performance. Moreover, the scheduling decision for each application component must be made before the component is executed making scheduling a *predictive* activity. A Grid scheduler, therefore, must be able to predict what the deliverable resource performance *will be* for the time period in which a particular application component will eventually use the resource.

The performance characteristics associated with a resource can be roughly categorized as either *static* characteristics or *dynamic* characteristics according to the speed with which they change. While the delineation can be rather arbitrary, static characteristics are ones that change slowly with respect to program execution lifetimes. For example, the clock-speed associated with a CPU is a relatively static (and quantifiable) performance metric. It is not completely invariant, however, as a given CPU may be replaced by one that is faster without changing other measurable characteristics. From the perspective of a Grid user, a CPU that has been upgraded to a faster clock-speed may look identical in terms of its other characteristics (memory size, operating system, etc.) before and after the upgrade.

Conversely, dynamic performance characteristics change relatively quickly. CPU loads and network throughput, for example, fluctuate with frequencies measured in minutes or seconds. It is such dynamic fluctuations that make Grid scheduling complex and difficult. Moreover, many studies have shown that the statistical properties associated with these performance fluctuations are difficult to model in a way that generates accurate predictions (Harchol-Balter and Downey, 1996; Crovella and Bestavros, 1997; Gribble et al., 1998; Harchol-Balter, 1999). The chief difficulties are either that the distribution of performance measurements can be modeled most effectively by a "power law" (i.e. the distribution is said to be "heavy-tailed") or that a time series of measurements for a given resource characteristic displays a slowly-decaying autocorrelation structure (e.g. the series is self-similar). Despite these statistical properties, however, users routinely make predictions of future performance levels based on observed history. For example, students in a University computer laboratory often use the advertised Unix load average as an indication of what the load will be for some time into the future.

Grid Resource Performance Prediction

Reconciling the theoretical difficulty associated with dynamic performance prediction with the practical observation that some kind of prediction is necessary in any scheduling context requires a careful formulation of the prediction problem. For Grid scheduling and resource allocation, two important characteristics can be exploited by a scheduler to overcome the complexities introduced by the dynamics of Grid performance response.

- **Observable Forecast Accuracy** — Predictions of future performance measurements can be evaluated dynamically by recording

the prediction accuracy once the predicted measurements are actually gathered.

- **Near-term Forecasting Epochs** — Grid schedulers can make their decisions dynamically, just before execution begins. Since forecast accuracy is likely to degrade as a function of time into the future for the epoch being forecast, making decisions at the last possible moment enhances prediction accuracy.

If performance measurements are being gathered from which scheduling decisions are to be made, predictions of future measurements can also be generated. By comparing these predictions to the measurements they predict when those measurements are eventually gathered, the scheduler can consider the quantitative accuracy of any given forecast as part of its scheduling decisions. Applications with performance response that is sensitive to inaccuracy can be scheduled using more stable resources by observing the degree to which those resources have been predictable in the past.

Grid schedulers can often make decisions just before application execution begins, and rescheduling decisions while an application is executing (either to support migration or for still-uncomputed work). Thus the time frame for which a prediction is necessary begins almost immediately after the scheduler finishes its decision-making process.

To exploit these characteristics, the forecasting infrastructure must, itself, be a high-performance, robust, long-running application. When scheduling decisions are made at run time, the time required to make the necessary forecasts and deliver them will be incurred as run time overhead. Hence, the forecasting system must be "fast" with respect to the application and scheduler execution times. If forecast information (even if it is only performance monitor data) that is not available because the system serving it has failed, "blind" and potentially performance-retarding decisions must be made. Finally, if forecast accuracy is to be considered as part of the scheduling process, the forecasting system must be constantly gathering historical measurement data and generating predictions from it.

In this chapter, we describe the functionality that is necessary to support resource allocation based on performance measurements taken from Grid resources. We detail our experiences with implementing this functionality as part of the Network Weather Service (NWS) (Wolski et al., 1999) — a distributed monitoring and forecasting service that is compatible with many different Grid infrastructures. We discuss the design and implementation decisions that are at the core of the system and outline its monitoring and forecasting capabilities. We conclude

with a discussion of future research and development challenges that must be overcome to make dynamic resource allocation more accessible to Grid programmers.

2. Grid Monitoring and Forecasting Infrastructure

Any system that is designed to support resource allocation based on performance data should provide three fundamental functionalities.

- **Monitoring**: Data from a distributed set of performance monitors must be gathered and managed so that it can be served.
- **Forecasting**: Resource allocators require forecasts of future performance. It is the forecast data (and not the monitor data) that ultimately must be served.
- **Reporting**: The information served by the system must be available in a wide-range of formats so that different scheduling and allocation implementations may be supported.

In Grid settings, these functionalities present some unique challenges.

Grid performance systems must be robust with respect to resource failure and/or restart, and must be carefully implemented so that their intrusiveness is minimized. Resource monitors, particularly those that probe resources by loading them, must be able to withstand frequent resource failure. Often, it is the faulty resources in a Grid that are of the most performance concern. If a monitor process requires manual intervention at restart, for example, there may be long periods of time for which no data is available precisely from the resources that must be most carefully monitored.

Moreover, if implemented as middleware, system administrators may not view performance monitoring and analysis systems as having the same level of importance as native operating system functionality. If and when a performance problem is reported by a user, the most frequent early response by many system administrators is to terminate any non-operating system monitoring processes for fear that they are cause of the observed difficulties. It is rare, however, for a monitor that has been prophylactically killed to be restarted once the true source of the performance problem is located. Therefore, middleware-based performance monitors must be self-reinitializing and they must be able to store the data that they produce in persistent storage that survives local intervention or system failure.

Another problem that Grid performance monitors must face stems from their use of the resources that they are monitoring. The intru-

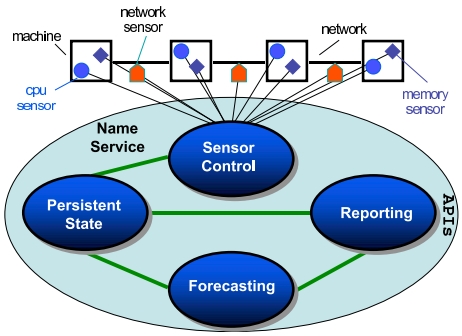

Figure 1.1. The Logical Architecture of the NWS

siveness of performance monitors, in terms of their induced CPU load, memory footprint, and storage footprint, constitutes system overhead and, thus, must be tightly controlled.

2.1 The Network Weather Service

The Network Weather Service (NWS) is a Grid monitoring and forecasting tool that has been designed to support dynamic resource allocation and scheduling. Figure 1.1 depicts its logical architecture in terms of independent subsystems. Sensors (typically independent processes) generate time-stamp, performance measurement pairs. For robustness and to limit intrusiveness, the system supports a sensor control subsystem that is distributed and replicated. Sensor processes can put control of their measurement cycle, sensor restart, etc. under the control of the NWS by adopting either a socket-based API, or an internal library API.

The NWS also assumes that performance sensors will be stateless, both to improve robustness and as a way of minimizing memory and storage footprints. To capture and preserve measurement data, the Persistent State subsystem exports a simple socket-based API that allows sensors to store their measurements remotely in time-series order. The number of Persistent State repositories, as well as the location and storage footprint of each are specifiable as installation parameters. In addition, new repositories can be added to the running system without reconfiguration.

Forecasts of future performance levels draw the historical data they require from the Persistent State system (and not the sensors). Thus, any process that can exercise the storage API exported by the Persistent State system, can inject measurements into the system for forecasting.

The forecasting subsystem is extensible, allowing the inclusion of new forecasting models into a forecaster library through a configuration-time API. To allow applications a way of trading off system complexity for performance, the NWS forecasting library can either be compiled into a Forecaster process and accessed remotely (thereby saving the local CPU and memory overhead) or loaded directly with the application.

To allow compatibility with a variety of Grid computing infrastructures, the NWS supports multiple reporting interfaces. These interfcaes communicate with the other subsystems via socket-based remote APIs as well, improving both flexibility and performance. New reporting formats can be added by providing a process or library that converts the NWS-internal API to the desired format.

In addition, this organization provides a convenient methodology for implementing replication and caching. Performance information (both measurement data and statistical forecasts) flow from the sensors, through the persistent state repositories and the forecasters to the reporting APIs, but not in the reverse direction. As such, reporting caches can be located near where the reports are consumed and can be replicated. Moreover by interrogating an internal Name Service (see below) the reporting caches can determine the frequency with which individual sensors are updating the various persistent state repositories. By doing so, each cache can refresh itself only when new data is expected from each sensor. When a scheduler or resource allocator queries a local cache, it receives up-to-date information without having to directly query the individual Persistent State repositories where the desired information is stored.

All components within an NWS installation register with an internal Name Service. The Name Service keeps track of the type, location (IP address and port number), and configuration parameters associated with each NWS process. In addition, all registrations are time limited and must be refreshed by their various components. Overall system status is determined by the active registrations that are contained within a given Name Service instantiation.

Under the current architecture, each instance of the Name Service defines a self-contained NWS installation. By using the name space to isolate separate NWS instantiations, multiple installations can overlay the same set of resources. Debugging or experimentation with alternative configurations (while a production version continues to run) is made easier by this design choice. At the same time, all of the components, including the sensors that are part of the distributed NWS release, run without privileged access. Thus, separate users can run individual instantiations of the NWS, each with its own Name Service.

2.2 The NWS Implementation

The engineering of a Grid performance system, particularly one designed to support resource allocation and scheduling, presents a unique challenge. In addition to the performance goals (response time and scalability) which are largely architectural issues, the implementation itself must be ubiquitous, robust, and non-intrusive. Ubiquity stems from two critical requirements: portability and the need to run with minimal privilege. Robustness and non-intrusiveness come, in part, from careful implementation techniques and extensive testing.

Any performance monitoring and forecasting system must be able to execute on *all* platforms available to the user. If a scheduler cannot "see" a system because no performance information is available, the system is for all intents and purposes not part of the Grid. This need is especially critical when a Grid is to be used to couple cheap, commodity resources with a unique instrument or machine. If the Grid infrastructure cannot execute on or monitor the unique instrument, the instrument cannot become part of a Grid.

To meet this need for ubiquity, the NWS is written primarily in C. At the time of this writing, it is the experience of the NWS implementation team that C is the most portable programming language. Most rare or unusual architectures support a C compiler and a subset of the Unix system calls. The NWS (with the exception of some of the sensor code) has been carefully coded to use only the most basic system services and generic ANSI C functionality. As a result, the core services have been quick to port to new systems as they become available. It is worth noting that the choice of C is not motivated, in this case, by performance but rather portability. The Java language environment is intended to provide the kind of portability the NWS requires. Many of the systems that users wish to access via a Grid, however, are large-scale machines with unique configurations. To date, the availability of a portable Java environment to machines of this class lags far behind the availability of C, if such a Java environment becomes available at all. At the same time, systems that do support a robust and standardized Java environment also support the baseline C functionality that is required by the NWS.

Figure 1.2 depicts the software organization of the system. The internal subsystems, the NWS-supplied sensors, the C and Unix command-line interface code are written in C. The HTML interface uses a combination of CGI and GNU tools (not distributed with the system) and the LDAP and SOAP interfaces are derived from open source software for implementing each protocol.

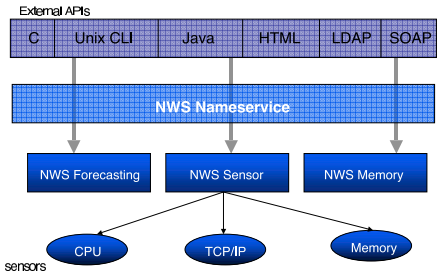

Figure 1.2. The Software Organization of the NWS Implementation

A second design decision alluded to earlier is that all NWS components must be able to run without privileged access. If an individual site wishes to configure a sensor that runs "as root," the extensibility of the system will permit it. Often, due to the security concerns associated with middleware, the need for privileged access tends to delay the deployment of a particular middleware component. Because the forecasting functionality is critical to resource allocation and scheduling, the NWS is coded to run with only minimal access privilege (e.g. a standard user login).

3. Performance Monitors

There are roughly two categories of performance monitor types: *passive* and *active*. A passive monitor is one which reads a measurement gathered through some other means (e.g. the local operating system). The best example of a passive monitor that most Grid systems report is the Unix Load Average metric. Almost all Unix and Linux systems (and their derivatives) record some measure of the number of jobs in the run queues of each processor on the machine. The frequency with which the queue length is sampled is operating system and operating system version specific. On most systems, however, a 1 minute, 5 minute, and 15 minute average of the run queue length are available although the way in which the average is calculated (arithmetic, geometric, exponentially smoothed, etc.) is again operating-system specific. This smoothed average of the run queue length defines the Load Average metric.

Systems such as the Globus Meta Directory Service (Czajkowski et al., 2001) report Unix Load Average by periodically querying the load average value and posting the result. Thus, the Globus load sensor passively

reads and reports a performance metric (Unix Load Average) that is gathered and maintained by the native operating system.

3.1 Intrusiveness versus Accuracy

The main advantage of passive sensing is that it is non-intrusive. The Unix Load Average is a measure that is already being generated. The sensor need only format and transmit the measured values appropriately. The difficulty with quantities such as Unix Load Average, however, is that they are sometimes complex to understand from a resource allocation perspective. For example, using load average as a measure of machine "busy-ness" allows machines of equivalent processing power to be ranked in terms of their expected execution speeds. The assertion that most Grid resource schedulers make is that in a pool of identical machines, the one with the smallest load average value is the one that will execute a sequential piece of code the fastest.

Using Unix load average to rank execution speeds implies that the presence of other jobs in each run queue will affect the performance of the scheduled application in the same way. Unix and Linux use an exponential aging algorithm to determine execution priority. Furthermore, the aging factor on some systems grows larger with occupancy time. The goal of this algorithm is to permit jobs that have recently completed an I/O operation to get the CPU immediately as an aid to response time. Consider interactive text editors as an example. After each key stroke, the editor is scheduled at a very high priority so that it can echo the character and then reblock waiting for the next key stroke. However, the priority aging algorithm rapidly lowers a processes priority to its set level if it does not immediately re-sleep after an I/O. Consider a system with a load average value of 2.0 where the two jobs are rapidly sleeping and waking. A CPU-bound Grid job sharing this system will get a different fraction of the CPU than on a system in which both jobs in the run queue are, themselves, CPU bound. In this latter case, the typical Unix scheduling algorithm degenerates into a round robin scheme. Thus, the load average implies a performance impact on a scheduled job that depends on the qualities of the other jobs that are running. This information, even it were published on a job-by-job basis, is difficult to interpret because it is the way in which jobs of different priorities interact that ultimately defines how load affects scheduling.

As an alternative method, a Grid performance monitor can periodically load the resource it is monitoring and record the observed performance response. This active approach has the advantage of disambiguating the relationship between a monitored quantity and performance im-

pact. Returning to the load average example, if a CPU monitor were to simply run a CPU bound process periodically, it could record the utilization that process enjoyed during each run. The fraction of wall-clock time that the process occupied the CPU can be used as the inverse of the slowdown caused by competing jobs on that system (e.g. a process getting 20% utilization can be thought of as 5 times slower than if it had received 100% utilization). The obvious difficulty with this approach is that the monitor must completely load the resource in order to measure it thereby leaving less resource available for actual computation.

There is an inherent tension between monitor accuracy and monitor intrusiveness that must be considered when designing a Grid performance sensor. The accuracy that active sensing makes possible must be balanced against the amount of resource it consumes. If good passive sensing techniques are available, it is sometimes possible to combine the two methods through some form of automatic regression technique.

As part of the Network Weather Service (NWS) Grid monitoring infrastructure, we have implemented a CPU sensor that combines Unix Load Average with active CPU probing. The sensor reads the 1 minute Load Average value periodically, according to a parameter set when the sensor is initialized. It also initiates a register-only CPU bound process (called the CPU probe) with a much lower periodicity and records the utilization that it experiences. The duration of the CPU probes execution is also a parameter. Anecdotally, we have found that a probe duration of 1.5 seconds is typically enough to yield accurate results.

Next, the sensor converts Unix Load Average to a utilization estimate. It assumes that the run queue will be serviced round-robin and that all jobs are CPU bound hence an equal fraction of time will be given to each. The sensor combines both the probe utilization and the Load Average reading by automatically calculating a bias value. If, for example, the utilization predicted by Load Average is 10% less than observed, the bias is computed as +10. Should the Load Average over-estimate utilization, the bias is negative.

The sensor reports as a measurement a utilization estimate that is generated by biasing the load average with the last bias recorded. Since load average is sampled much more frequently than the probe is run, the intrusiveness is less than if only the probe were used. At the same time, the probe captures some of the interaction between itself and other contending jobs in the run queue.

Finally, the NWS CPU sensor controls the periodicity with which the probe is executed based on the changing size of the bias. If the bias value is fluctuating the sensor assumes that the load is highly fluctuating and the CPU should be probed again in a relatively short period of time. If

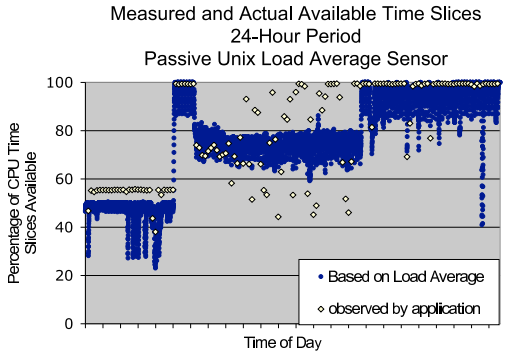

Figure 1.3. A Comparison of Available CPU Cycles as Measured with Unix Load Average to Actual Observed Occupancy Percentage

the bias is relatively stable, the probe frequency is decreased. Both the maximum and minimum frequencies as well as the stability threshold are configuration parameters to the sensor.

Figures 1.3 and 1.4 depict the effects of this sensing technique using a workstation as an example. In Figure 1.3, the solid circles show the percentage of available CPU time slices (over a 10 second period) that are measured by Unix Load Average. The $y-axis$ values are measurements, and the $x-axis$ values show time of day. One measurement occurs at every 10 second interval, and the total trace covers a 24-hour period. This particular workstation was being used by a graduate student at the time to finish her Ph.D. thesis, making the load variation (however non-synthetic) potentially atypical.

To convert a load average measurement to an available occupancy percentage, the NWS passive sensor uses the formula

$$load\_avg\_available\_cpu = 100.0/(load\_average + 1.0) \qquad (1.1)$$

where the load average covers 1 minute. Again, based on the assumption that all processes in the run queue have equal priority, the available fraction of CPU time for a new process is 1 divided by the number of currently runnable processes plus an additional process. Multiplying by 100 simply converts this number into a percentage.

Diamond shapes (drawn in outline with a light-colored fill) show the occupancy observed by a test program that occurs at less frequent intervals (every 10 minutes) in the trace. When executed, the test program spins in a tight loop for 30 seconds, measured in wall-clock time, and records the user and system occupancy time during the execution. The ratio of actual occupancy time to wall-clock time is the observed availability fraction. Both the 10 minute interval, and the 30 second

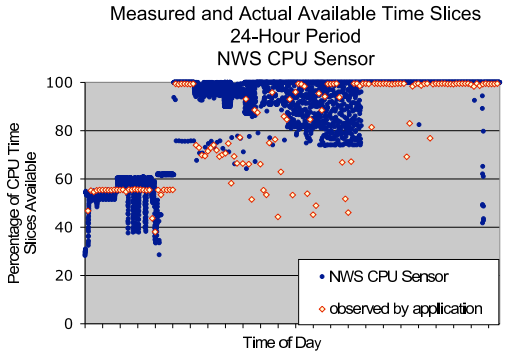

Figure 1.4. A Comparison of Available CPU Cycles as Measured with NWS CPU Sensor to Actual Observed Occupancy Percentage

execution duration allow the smoothed load average value to "recover" from the load introduced by the test program. During the measurement period, the test program and the load average sensor were coordinated so that a load average measurement was taken immediately before each test program run, and both were assigned the same time stamp. Thus the vertical distance between each light colored diamond and corresponding solid circle in the figure shows graphically the measurement error associated with each measurement.

Figure 1.4 shows the same accuracy comparison for the NWS CPU sensor. In it, each solid circle represents the biased NWS sensor value and, as in the previous figure, each light-colored diamond shows the occupancy observed by the test program. By learning and then applying a bias value, the NWS sensor is better able to measure the true availability experienced by the test application with little added intrusiveness.

More generally, however, this example illustrates the need for Grid resource monitoring systems to capture measurement error. Many such systems report the metrics that are available to users (e.g. Unix Load Average) but few provide estimates of how those measurements translate into observable application performance. For resource allocation and scheduling purposes, the measurement error associated with passive measurements is a useful and often overlooked quantity.

3.2 Intrusiveness versus Scalability

Another important design point concerns the trade-off between intrusiveness and scalability. Consider the problem of gathering periodic end-to-end network probe information. The naive implementation furnishes each sensor with a list of other sensors to contact within a Grid,

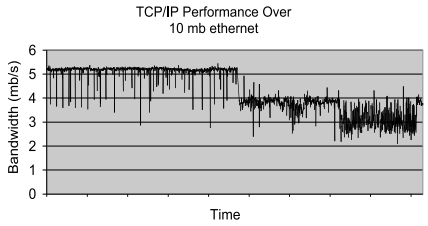

Figure 1.5. TCP/IP Sensor Contention

and a periodicity. Each sensor operates on its own clock and with the specified periodicity probes all of the other sensors.

In Figure 1.5 we show a network performance time series of the TCP/IP performance observed between a pair of Unix hosts connected via 10 megabit-per-second Ethernet. Each bandwidth reading is generated by timing a 64 kilobyte transfer using a TCP/IP socket with 32 kilobyte socket buffers. During the first half of the trace (the left side of the figure) only one pair of hosts — a sender and a receiver — was probing the network. Midway through the trace, a second host pair began to probe the network simultaneously. The loss of available bandwidth, which is visually apparent from the trace, results from the interaction of colliding network probes.

To produce a complete end-to-end picture of network performance between N hosts, $2 * (N^2 - N)$ such measurements would be required (i.e. one in each direction and hosts do not probe themselves). If each host uses its own local clock to determine when to probe the network, the likelihood of probe contention goes up at least quadratically as the Grid scales.

To prevent probe contention, the NWS end-to-end network sensor uses a token-passing protocol to implement mutual exclusion between "cliques" of hosts. Hosts within a specified clique pass the entire clique list in a token. The NWS clique protocol implements a simplified leader election scheme that manages token loss/recovery, and network partitioning. If a token is lost because the host hold it fails, the other hosts in the clique will time out and attempt to elect themselves leader by regenerating and sending out a new token. Time stamps on the token are used to resolve the possibility of multiple simultaneous time outs. When a host encounters two different tokens (from two different leaders) it will

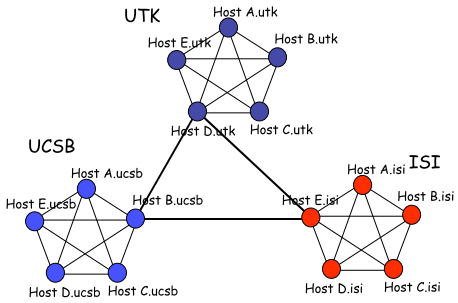

Figure 1.6. Example NWS Clique Hierarchy

"kill" the older one. This scheme also manages network partitioning. If the network partitions, the hosts that are separated from the current leader will elect a new leader on their side of the partition. When the partition is resolved, the two tokens will once again circulate across the entire host list, and one of them (the older one) will be annihilated. This form of active replication makes the clique protocol robust to both host and network failure.

To permit extensibility and scalability, sensors can participate in multiple cliques at the same time, and each clique can contain any number of hosts greater than or equal to 2. Thus, the clique organization can capture a variety of non-clique monitoring topologies if probe contention is not a concern. For example, one common topology that many sites wish to monitor is a "star" topology: one distinguished host connected to a set of satellite hosts, without connectivity between the satellites. If probe contention is not an issue, one clique consisting of a satellite node and the central node can be created for each satellite node. Since the central node participates in multiple cliques simultaneously, this organization implements the desired measurement topology. This, the NWS clique abstraction can be used to implement other monitoring topologies according to the needs of each individual installation.

To gain scalability, cliques can be organized into a hierarchy. At the bottom level of the hierarchy are cliques of hosts. Each clique "promotes" a distinguished representative to participate in a higher-level clique, forming a tree. Consider the example shown in Figure 1.6. In it, five hosts (labeled A, B, C, D, and E) are configured into "base" cliques at each of three sites: UCSB, ISI, and UTK. One distinguished host from each site participates in a higher-level clique that captures inter-site connectivity.

Notice that this organization can capture the full N^2 matrix of connectivity if the inter-site connectivity performance is similar for all nodes communicating between sites. For example, if UCSB is the University of California in Santa Barbara, and UTK is the University of Tennessee, in Knoxville, any host in the UCSB clique communicating with any host in the UTK clique will likely observe the same network performance since much of the network between the two will be shared. That is, since virtually all UCSB-to-UTK network traffic will traverse common network elements, a single UCSB-UTK pair can measure the inter-site connectivity. By using the inter-site measurements of the distinguished pair in the higher-level clique in place of the missing measurements, the NWS can construct a full N^2 picture without conducting N^2 measurements. At the same time, measurements within each clique will not contend.

4. Forecasting

The problem of determining a resource allocation or schedule that maximizes some objective function is inherently a predictive one. When a decision is made about the resources to allocate, some assumption about the future behavior of the resources or application is either implicitly or explicitly included. For example, if a large MPI program is to be assign to a parallel machine because of the processor speeds, memory capacity, and interconnect speed, the scheduler or resource allocator making that decision is making an assumption of what the processor speeds, memory availability, and interconnect speed *will be* when the MPI program runs. Users of space-shared parallel machines often assume that these predictions are easy to make statically. Interconnect speed, however, may be influenced by parallel jobs running in other partitions, so even in traditional parallel computing settings, predictions of dynamically changing behavior may be required.

In Grid settings, however, where resources are federated and interconnected by shared networks, the available resource performance can fluctuate dramatically. The same scheduling decisions based on predictions of future resource performance are necessary if application are to obtain the performance levels desired by their users. Therefore, some methodology is required to make forecasts of future performance levels that upon which scheduling decisions can be based. Note that even though we have outlined the need for forecasting explicitly, all Grid users go through this activity either explicitly or implicitly. When a user chooses a particular data repository, for example, because it is connected to a "faster" network, he or she is making the prediction that the network *will be* faster when it is used to access the user's data. Most often this

forecast is based on past experience with the resource. The NWS includes statistical methods that attempt to mechanize and automate the forecasting process for the user based on similar historical experience.

4.1 The NWS Non-Parametric Forecasting Method

The principle behind the NWS forecasting technique is that the best forecasting technique amongst a number of available options can be determined from past accuracy. Each forecasting method is configured into the system with its own parameters. It must be able to generate, on demand, a prediction based on a previous history of measurements and forecasts. That is, for each forecasting method f at measurement time t,

$$prediction_f(t) \quad = \quad METHOD_f(history_f(t)) \tag{1.2}$$

where

$prediction_f(t)$ = the predicted value made by method f for the measurement value at $t+1$,
$history_f(t)$ = a finite history of measurements, forecasts, and forecast errors generated previously to time t using method f, and
$METHOD_f$ = forecasting method f.

Each method is presented with a history of previous measurements (represented as a time series) and maintains its own history of previous predictions and accuracy information. In particular,

$$err_f(t) = value(t) - prediction_f(t-1) \tag{1.3}$$

is the error residual associated with a measurement $value(t)$ taken at time t and a prediction of that measurement generated by method f generated at time $t-1$.

The *primary forecasters* are able to produce a forecast based on time-series data and some set of parameters. The forecaster interface is general enough to accept a variety of forecasting techniques. Because the autocorrelation structure of many performance series is complex, and because series stationarity is unlikely, a large set of fast, simple predictors that can be constantly re-evaluated is the most effective configuration.

The primary techniques to produce forecasts include mean-based and median-based methods for producing completely non-parametric forecasts. Based on the observation that more recent data is often more indicative of current conditions, the primary forecasters make use of varying amounts of history using "sliding window" techniques. In this same

spirit exponential smoothing techniques, parameterized by the amount of *gain*, are used to produce forecasts as well. Each of these forecasting modules accepts data as a time-series and computes a forecast from that data and any parameters that the module accepts.

4.2 Secondary Forecasters: Dynamic Predictor Selection

The NWS operates all of the primary forecasters (and their various parameterizations) simultaneously at the time a forecast is requested through the API. It then uses the error measure calculated in Equation 1.3 to produce an overall fitness metric for each method. The method exhibiting the lowest cumulative error at time t is used to generate a forecast for the measurement at time $t+1$ and that forecast is recorded as the "winning" primary forecast. The NWS conducts two such error tournaments for each forecast: one based on the mean square error and one based on the mean absolute error.

$$MSE_f(t) = \frac{1}{t+1}\sum_{i=0}^{t}(err_f(i))^2 \tag{1.4}$$

and the mean absolute prediction error

$$MPE_f(t) = \frac{1}{t+1}\sum_{i=0}^{t}|(err_f(i)| \tag{1.5}$$

We then define

$$MIN\_MSE(t) = predictor_f(t) \quad \text{if } MSE_f(t) \text{ is the minimum over all methods at time } t \tag{1.6}$$

and

$$MIN\_MAE(t) = predictor_f(t) \quad \text{if } MAE_f(t) \text{ is the minimum over all methods at time } t. \tag{1.7}$$

That is, at time t, the method yielding the lowest mean square prediction error is recorded as a forecast of the next measurement by MIN_MSE. Similarly, the forecasting method at time t yielding the lowest overall mean absolute prediction error becomes the MIN_MAE forecast of the next measurement.

Both of these error metrics use the cumulative error spanning the entire history available from the series. In an attempt to address the possibility that the series is non-stationary, the system also maintains error-minimum predictors where the error is recorded over a limited

previous history. The goal of this approach is to prevent error recordings that span a change point in the series from polluting the more recent error performance of each predictor. More formally

$$MIN\_MSE_W(t,w) = predictor_f(t) \quad \text{if } MSE_f(t) \text{ is the minimum over all methods at time } t \text{ for the most recent w measurements} \tag{1.8}$$

and

$$MIN\_MAE_W(t,w) = predictor_f(t) \quad \text{if } MAE_f(t) \text{ is the minimum over all methods at time } t \text{ for the most recent w measurements} \tag{1.9}$$

where w is a fixed window of previous measurements.

4.3 The NWS Forecast

Finally, the NWS forecast that is generated on demand is the one that "wins" an error tournament at time t for the set of primary and secondary forecasters described in this section. That is

$$NWS\_MSE(t) = predictor_f(t) \quad \text{if } MSE_f(t) \text{ is the minimum over all primary and secondary methods at time } t \tag{1.10}$$

and

$$NWS\_MAE_W(t) = predictor_f(t) \quad \text{if } MAE_f(t) \text{ is the minimum over all primary and secondary methods at time } t. \tag{1.11}$$

Table 1.1 summarizes the primary and secondary methods that are the NWS uses for analysis. The NWS combines these methods to produce an NWS_MSE and NWS_MAE forecast for each value in each series presented to the forecasting subsystem. Because many of the primary forecasters can be implemented using computationally efficient algorithms, the overall execution cost of computing the final NWS forecasts is low. For example, on a 750 megahertz Pentium III laptop, each NWS_MSE and NWS_MAE requires 161 microseconds to compute.

Table 1.1 summarizes the primary and secondary methods that are the NWS uses for analysis.

5. Conclusions, Current Status, Future Work

The heterogeneous and dynamic nature of Grid resource performance makes effective resource allocation and scheduling critical to application

Predictor	**Description**	**Parameters**
$LAST$	last measurement	
RUN_AVG	running average	
EXP_{SM}	exponential smoothing	$g = 0.05$
EXP_{SM}	exponential smoothing	$g = 0.10$
EXP_{SM}	exponential smoothing	$g = 0.15$
EXP_{SM}	exponential smoothing	$g = 0.20$
EXP_{SM}	exponential smoothing	$g = 0.30$
EXP_{SM}	exponential smoothing	$g = 0.40$
EXP_{SM}	exponential smoothing	$g = 0.50$
EXP_{SM}	exponential smoothing	$g = 0.75$
EXP_{SM}	exponential smoothing	$g = 0.90$
EXP_{SMT}	exponential smooth+trend	$g = 0.05, tg = 0.001$
EXP_{SMT}	exponential smooth+trend	$g = 0.10, tg = 0.001$
EXP_{SMT}	exponential smooth+trend	$g = 0.15, tg = 0.001$
EXP_{SMT}	exponential smooth+trend	$g = 0.20, tg = 0.001$
$MEDIAN$	median filter	$K = 31$
$MEDIAN$	median filter	$K = 5$
SW_AVG	sliding window avg.	$K = 31$
SW_AVG	sliding window avg.	$K = 5$
$TRIM\_MEAN$	α-trimmed mean	$K = 31, \alpha = 0.3$
$TRIM\_MEAN$	α-trimmed mean	$K = 51, \alpha = 0.3$
$ADAPT\_MED$	adaptive window median	$max = 21,\ min = 5$
$ADAPT\_MED$	adaptive window median	$max = 51,\ min = 21$
MIN_MSE	adaptive minimum MSE	
MIN_MAE	adaptive minimum MAE	
MIN_MSE_W	windowed adaptive minimum MSE	$w = 1$
MIN_MAE_W	windowed adaptive minimum MAE	$w = 1$
MIN_MSE_W	windowed adaptive minimum MSE	$w = 5$
MIN_MAE_W	windowed adaptive minimum MAE	$w = 5$
MIN_MSE_W	windowed adaptive minimum MSE	$w = 10$
MIN_MAE_W	windowed adaptive minimum MAE	$w = 10$
MIN_MSE_W	windowed adaptive minimum MSE	$w = 30$
MIN_MAE_W	windowed adaptive minimum MAE	$w = 30$
MIN_MSE_W	windowed adaptive minimum MSE	$w = 50$
MIN_MAE_W	windowed adaptive minimum MAE	$w = 50$
MIN_MSE_W	windowed adaptive minimum MSE	$w = 100$
MIN_MAE_W	windowed adaptive minimum MAE	$w = 100$

Table 1.1. Summary of Forecasting Methods

performance. The basis for these critical scheduling functionalities is is a predictive capability that captures future expected resource behavior. Typically, Grid users and schedulers will use the immediate performance history (the last observed value or a running average) to make an implicit prediction of future performance. However, there are several important ways in which such an *ad hoc* methodology can be improved.

To be truly effective, the performance gathering system must be robust, portable, and non-intrusive. Simply relying on available resource performance measurements, or building naive probing mechanisms can result in additional resource contention and a substantial loss of application performance. Moreover, by carefully considering measurement error, it is possible to automatically and adaptively balance the accuracy of explicit resource probing with the non-intrusiveness of passive measurement. Similarly, overhead introduced by the performance gathering system must be explicitly controlled, particularly if probes can contend for resources. The ability to implement this control in a way that scales with the number of resources requires an effective system architecture for the performance monitoring system.

By using fast, robust time-series techniques, and running tabulations of forecast error, it is possible to improve the accuracy of performance predictions with minimal computational complexity. In addition to point-valued predictions, these same adaptive techniques can generate empirical confidence intervals and automatic resource classifications thereby improving scheduler design and scalability.

Thus, effective support for dynamic resource allocation and scheduling requires an architecture, a set of analysis techniques, and an implementation strategy that combine to meet the demands of the Grid paradigm. The Network Weather Service has been developed with these realizations in mind. It is a robust, portable, and adaptive distributed system for gathering historical performance data, making on-line forecasts from the data it gathers, and disseminating the values it collects.

5.1 Status

Currently, the NWS is available as a released and supported Grid middleware system from the National partnership for Advanced Computational Infrastructure (NPACI) and from the National Science Foundation's Middleware Initiative (NMI) public distribution. These distributions include portable CPU, TCP/IP socket sensors, a non-paged memory sensor for Linux systems, and support for C, Unix, HTML, and LDAP interfaces (the latter via a caching proxy). In addition, the NWS team distributes a non-supported version with additional proto-

type functionalities that have yet to make it into public release. At the time of this writing, the working prototypes include an NFS probing file system sensor, a portable non-paged memory sensor, an I/O sensor, and a senor that monitors system availability. There is also a prototype Open Grid Systems Architecture (phy,) interface and a Microsoft .NET/C# implementation as well.

5.2 Future Work

NWS development will expand the system's utility in three ways. First, we are investigating new statistical techniques that enable more accurate predictions than the system currently generates. While the NWS forecasts are able to generate useful predictions from difficult series, the optimal postcast measures indicate that more accurate forecasts are still possible. We are also studying ways to predict performance characteristics that do not conform well to the time series model. Periodic measurement is difficult to ensure in all settings and a key feature of the NWS is its ability to cover the space of Grid forecasting needs. Secondly, we are exploring new reporting and data management strategies such as caching OGSA proxies and relational data archives. These new data delivery systems are needed to support an expanding user community, some of whom wish to use the NWS for statistical modeling as well as resource allocation. Finally, we are considering new architectural features that will enable the system to serve work in peer-to-peer settings where resource availability and dynamism are even more prevalent than in a Grid context.

For Grid resource allocation and scheduling, however, the NWS implements the functionalities that are necessary to achieve tenets of the Grid computing paradigm (Foster and Kesselman, 1998) with the efficiency that application users demand. While we have outlined the requirements for Grid performance data management in terms of the NWS design, we believe that these requirements are fundamental to the Grid itself. As such, any truly effective resource allocation and scheduling system will need the functionality that we have described herein, independent of the way in which the NWS implements this functionality. Thus, for the Grid to be a success, effective forecasts of resource performance upon which scheduling and allocation decisions will be made, are critical.

References

The physiology of the grid: An open grid services architecture for distributed systems integration. http://www.gridforum.org/ogsa-wg/.

Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Dennis Gannon, L. J., Kennedy, K., Kesselman, C., Reed, D., Torczon, L., and Wolski, R. (2001). The grads project: Software support for high-level grid application development. *International Journal of High-performance Computing Applications*, 15(4):XXX.

Berman, F., Wolski, R., Figueira, S., Schopf, J., and Shao, G. (1996). Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*. (Pittsburgh).

Crovella, M. and Bestavros, A. (1997). Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5:XXX.

Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C. (2001). Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE 5'gmposium on High-Performance Distributed Computing.*

Foster, I. and Kesselman, C., editors (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc.

Gribble, S. D., Manku, G. S., Roselli, D., Brewer, E. A., Gibson, T. J., and Miller, E. L. (1998). Self-similarity in file systems. In *Proceedings of SIGMETRICS '98.*

Harchol-Balter, M. (1999). The effect of heavy-tailed job size distributions on computer system design. In *Proceedings of ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics, Engineering and Statistics.*

Harchol-Balter, M. and Downey, A. (1996). Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.*

Petitet, A., Blackford, S., Dongarra, J., Ellis, B., Fagg, G., Roche, K., and Vadhiyar., S. (2001). Numerical libraries and the grid. In *Proceedings of IEEE SC'01 Conference on High-performance Computing.*

Ripeanu, M., Iamnitchi, A., and Foster., I. (2001). Cactus application: Performance predictions in a grid environment. In *Proceedings of European Conference on Parallel Computing (EuroPar).*

Spring, N. and Wolski., R. (1998). Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing.*

Wolski, R., Spring, N., and Hayes, J. (1999). The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6).