The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing

Rich Wolski [a,1] Neil T. Spring [b,2] Jim Hayes [b,3]

[a] *University of California, San Diego and University of Tennessee, Knoxville*
[b] *University of California, San Diego*

Abstract

The goal of the Network Weather Service is to provide accurate forecasts of dynamically changing performance characteristics from a distributed set of metacomputing resources. Providing a ubiquitous service that can both track dynamic performance changes and remain stable in spite of them requires adaptive programming techniques, an architectural design that supports extensibility, and internal abstractions that can be implemented efficiently and portably. In this paper, we describe the current implementation of the NWS for Unix and TCP/IP sockets and provide examples of its performance monitoring and forecasting capabilities.

Key words: network weather; network monitoring; performance prediction; metacomputing; network-aware; distributed computing

1 Introduction

Increasingly, high-quality networks have made it possible for users to employ widely dispersed computational and data resources. While the pervasiveness of the world wide web illustrates one obvious example, almost all computational constituencies have come to expect that some form of network connectivity will be attached to **all** potentially useful resources.

[1] Supported by DARPA N66001-97-C-8531 and NSF ASC-9701333
[2] Supported by Department of Defense Modernization Program (NAVO)
[3] Supported by NPACI (NSF)

With ubiquitous connectivity comes the ability to choose between otherwise equivalent resources based on their perceived performance. For example, students sharing resources in the computer science department at UCSD frequently try to choose the most lightly loaded server for their activities. However, it is not the current load on each system that interests them but, rather, an estimate of what the load **will be** in the near future when they execute a program. They use information available from the system (e.g. Unix *load average*, the number of users currently logged in, who those users are, what programs are currently running, etc.) to **predict** what performance will be delivered to their program.

In this paper we describe the latest implementation of the **Network Weather Service** (NWS), a distributed, generalized system for producing short-term performance forecasts based on historical performance measurement. The goal of the system is to dynamically characterize and forecast the performance deliverable at the application level from a set of network and computational resources. Such forecasts have been used successfully to implement dynamic scheduling agents for metacomputing applications [26,3], and to choose between replicated web pages [1].

Implementing the NWS to operate in a variety of metacomputing and distributed environments, each with its own dynamically changing performance characteristics, has illuminated the utility of adaptive programming techniques, distributed fault-tolerant control algorithms, and an extensible system architecture. We focus on the role of these methodologies in building and deploying the system.

The next section discusses the design goals for, and the overall architecture of, the current NWS implementation. The remaining sections of the paper describe the function of the system's core component processes and how they combine to form a generalizable service for managing dynamically changing performance information. Section 3 describes naming and persistent state management that the system uses internally, Section 4 describes performance monitoring facilities used by the system, Section 5 describes the forecasting features that are currently supported, and Section 6 details the programming and web interfaces that are available. In Section 7, we focus on some of the adaptive control and fault-tolerance mechanisms implemented throughout the system. Section 8 discusses related efforts, and in Section 9 we summarize the work and discuss future research goals.

2 System Architecture

The design of previous NWS implementations [31,30,29] focused on providing the functionality necessary to investigate the effectiveness of dynamic scheduling in local, medium, and wide-area computational settings [26,3]. These implementations did not scale well, however, and lacked the robustness necessary to make the NWS a reliable system service. Moreover, we wished to extend the monitoring and forecasting capabilities of the system to meet the needs of various performance-oriented distributed software infrastructures such as Globus [12], Legion [18], Condor [27] and Netsolve [7]. As such, we hoped to improve the portability, the extensibility, and the reliability of the system over prior implementations.

The NWS is designed to maximize four possibly conflicting functional characteristics. It must meet these goals despite the highly dynamic execution environment and evolving software infrastructure provided by shared metacomputing systems [2].

- **Predictive Accuracy**: The NWS must be able to provide accurate estimations of future resource performance in a timely manner.
- **Non-intrusiveness**: The system must load the resources it is monitoring as little as possible.
- **Execution longevity**: To be effective, the NWS should be available at any time as a general system service. It should not execute and complete – its execution lifetime is logically indefinite.
- **Ubiquity**: As a system service, the NWS should be available from all potential execution sites within a resource set. Similarly, it should be able to monitor and forecast the performance of all available resources.

We have constructed the current NWS using using four different *component processes*.

- **Persistent State** process: stores and retrieves measurements from persistent storage.
- **Name Server** process: implements a directory capability used to bind process and data names with low-level contact information (e.g. TCP/IP port number, address pairs).
- **Sensor** process: gathers performance measurements from a specified resource.
- **Forecaster** process: produces a predicted value of deliverable performance during a specified time frame for a specified resource.

Each of these component processes, represented in Figure 1, can communicate with other processes only through strongly typed messages. Their implementation may be improved or replaced by appropriate standard implementations

as they become available.

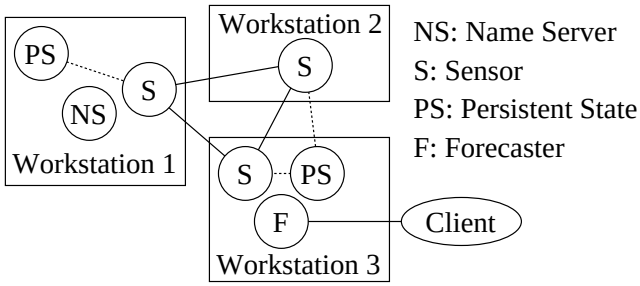

Fig. 1. NWS Processes distributed across three workstations. The Name Server resides on only one host in the system. Sensors monitor the performance characteristics of networks and processors and send their measurements to Persistent State managers. The Forecaster acts as a proxy for application scheduling clients and user queries. Workstation 2 can be integrated in the system without any associated storage space, since its persistent state is managed on Workstation 3.

At present, each of the five process abstractions has been implemented in C, for Unix, using TCP/IP socket networking facilities as the underlying communications transport. Our choice of Unix and TCP/IP sockets as an initial programming platform stems from their nearly exclusive use by extant metacomputing infrastructures such as Globus [12] and Condor [27]. The remainder of this paper, therefore, focuses on the implementation of the NWS for Unix and Unix networking via sockets.

3 Naming and State Management

To make the system more robust, all NWS processes are stateless. Persistent state – state that must be able to survive the failure of a process' memory – is managed explicitly throughout the system using Persistent State processes. Each Persistent State process provides a simple text string storage and retrieval service and allows each stored string to be associated with an optional time stamp. Each storage or retrieval request must be accompanied by the name of the data set that is to be accessed, and any data that is sent to a Persistent State process is immediately written to disk before an acknowledgement is returned. Since the function of the NWS is to generate forecasts which lose their utility after their epoch passes, the system does not maintain any data indefinitely. Each file that a Persistent State process uses is managed as a circular queue, the length of which is a configuration option. Data to be archived indefinitely must be fetched and stored in some more permanent medium outside the NWS before the queue fills.

The NWS also maintains its own primitive but highly portable naming and directory service to manage name-location bindings. In the current implemen-

tation, a name is a human-readable text string, and a location is a TCP/IP address and port number, but all data are manipulated as text strings. At present, the Name Server process that implements this functionality is based on the more general Persistent State process. This relationship is purely an engineering expediency, however, as the circular queue management techniques implemented for Persistent State storage are cumbersome to use to implement a directory service. We are, therefore, converting the Name Service to use an implementation of the Lightweight Directory Access Protocol [32] (LDAP).

The address of the NWS Name Server process is the only well-known address used by the system, allowing both data and services to be distributed [4]. All other NWS processes register their name-location bindings with the Name Server. These bindings time out according to a time-to-live specification that must accompany each registration. Active processes, therefore, must register their bindings periodically. This approach provides a simple "heartbeat" that is process specific. We are considering the use of the Globus Heartbeat Monitor [13] as an implementation platform for this functionality as part of our future development.

We anticipate that state storage and name service functionality will eventually be provided by lower-level metacomputing services, such as the Globus Metacomputing Directory Service [10] and the Legion Resource Directory Service [8].

4 Performance Monitoring

The problems associated with gathering accurate performance measurements from active computational and network resources continue to pose significant research challenges [24,21,5,6,20,8,19]. In general, there is a tension between the intrusiveness of a monitoring technique and the measurement accuracy it provides. The NWS attempts to use both extant performance monitoring utilities and active resource occupancy to measure performance. The current implementation supports measuring the fraction of CPU time available for new processes, TCP connection time, end-to-end TCP network latency, and end-to-end TCP network bandwidth.

[4] At present, the Name Server is centralized, but we plan to leverage the distribution facilities of LDAP once they become available.

4.1 NWS Sensors

The function of an NWS *Sensor* is to gather and store time stamp-performance measurement pairs for a specific resource. Each Sensor process may measure several different performance characteristics of the resource it is sensing. The TCP/IP network Sensor, for example, provides both bandwidth and end-to-end round-trip latency measurements, but each set of measurements is named and stored separately. That is, a Sensor does not attempt to correlate the separate performance characteristics of a resource it monitors. However, since a Sensor attaches a time stamp to each measurement it takes, different types of measurements may be associated by matching their time stamps. While we have implemented several different Sensors (described below) any process that can generate time stamp-measurement tuples, store them with a Persistent State process, and register their location with a Name Server process can contribute data to the system.

4.2 CPU Sensor

The NWS CPU Sensor combines information from Unix system utilities *uptime* and *vmstat* with periodic active CPU occupancy tests to provide measurements of CPU availability on timeshared Unix systems. CPU availability is measured as the fraction of CPU occupancy time a full-priority standard user process can obtain. Typically, the *uptime* utility reports load average as the average number of processes in the run queue over the past one, five, and fifteen minutes. The CPU Sensor uses the one-minute measurement to calculate the fraction of the CPU occupancy time that a process would get if it were to run at the moment the *uptime* measurement were taken. From *vmstat* output, the CPU Sensor uses a combination of idle time, user time, and system time measurements to generate an estimate of the available CPU occupancy fraction [31].

Since these utilities generate their reports using internal Unix system variables, invoking the utilities presumably does not generate significant load. That is, they are fairly non-intrusive monitoring utilities. However, both may leave out considerable information that can affect measurement of CPU availability. For example, neither *uptime* nor *vmstat* provides information on the priority of processes presently running on the system. In order to obtain more accurate measurements, the CPU Sensor incorporates active probes into its calculations. It periodically runs an artificial, compute-intensive "probe" program and calculates the CPU availability as the ratio of its observed CPU occupancy time to the wall-clock time of its execution. The Sensor then compares the results of these probes to the measurements taken using *uptime* and

vmstat and uses the utility that is reporting the most accurate information. Typically, the probe process is run much less frequently than measurements are gathered from *vmstat* or *uptime*. When all three values are taken simultaneously, the probe is treated as "ground truth" and used to bias subsequent *uptime* and *vmstat* measurements until the next probe is conducted.

The Sensor also uses heuristics to adaptively adjust the frequency with which active probes are conducted, thereby further limiting its intrusiveness. As long as the measurements from *uptime* and *vmstat* remain relatively stable, the Sensor assumes that the error in the estimate will also change little, and so runs active experiments less frequently. On systems with very stable usage patterns the Sensors may run active probes only once per hour. Conversely, when utility estimates change significantly between sequential measurements, the Sensor increases the frequency of active probes in order to calculate more accurate error estimates.

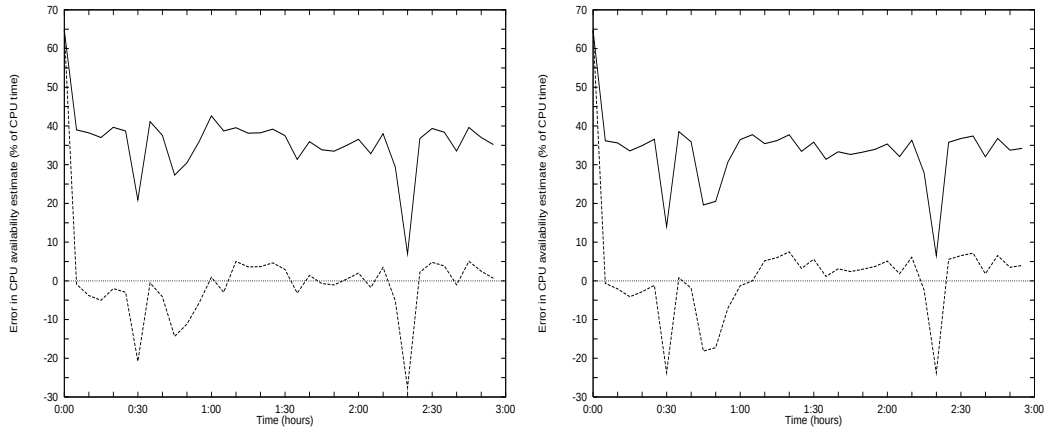

Fig. 2. Improvement from active probing in estimates of CPU availability generated using *uptime* (left) and *vmstat* (right). The solid line shows the amount of error in unadjusted estimates; the dashed line the error in adjusted estimates.

Figure 2 shows an example of improvement in CPU availability estimates by incorporating infrequent runs of an active probe. CPU availability estimates were generated on a UCSD Parallel Computing Lab workstation by an NWS Sensor over a three-hour period, during which time a low-priority process was running. Because neither *uptime* nor *vmstat* returns priority information, unadjusted estimates of CPU availability, shown by the solid lines, differ from the actual values by as much as 40% of CPU time. Adjusting the estimates using the results of active probing gives the improved estimates shown by the dashed lines. During the three-hour test, the NWS sensor ran the three-second active probe seven times, consuming less than 0.2% of the overall CPU time in order to provide the improved estimates.

4.3 Network Sensor

Because end-to-end network performance data between arbitrary machines is not consistently available, NWS network Sensors rely on active network probes exclusively when determining network load. Each probe consists of a timed network operation, such as the movement of a fixed amount of data, or, in the case of TCP, the establishment and dissolution of a network connection. At regular intervals, each network Sensor connects to a set of peer Sensors running on machines of interest and conducts one or more probes of different types to gather its measurements. To gather a set of end-to-end performance measurements of any type from N Sensors would require $N^2 - N$ probes. To avoid introducing this much network load, Sensors are organized hierarchically so that an end-to-end measurement can be made for a representative subset of the total Sensor population. These representative measurements can then be used to describe the network performance between an arbitrary Sensor pair. We discuss this hierarchical organization in greater detail in Section 7.

Currently, the NWS network Sensor is capable of measuring three network performance characteristics: small-message round-trip time, large-message throughput, and TCP socket connect-disconnect time. The small-message probe consists of a 4-byte TCP socket transfer that is timed as it is sent from a source Sensor to a destination Sensor and back. The socket connection used to facilitate the transfer is already established before the probe is conducted. Large-message throughput (that is taken to measure available network bandwidth at the application level) is calculated by timing the transfer of a message using TCP and the acknowledgement of its receipt by the receiving sensor. The size of the message, the sending and receiving socket buffer requests, and the size of the internal buffers used by each Sensor in the socket `send()` and `recv()` system calls are all parameterizable for each Sensor. Empirically, we have observed that a message size of 64K bytes, sent using 32K byte socket buffers and 16K byte `send()` and `recv()` yields meaningful results.

It is important to note, however, that the network performance to the application level can be affected dramatically by socket interface conditioning. The vBNS [28] for example, supports high throughput rates, both in aggregate and end-to-end, if the RFC1323 large-window extensions are used to condition the sockets used in the transfer. In our experience, however, most "standard" socket communications do not use these extensions at the time of this writing. Therefore, we have elected not to use them when the network Sensor measures deliverable network performance. We do plan to extend the system by developing a large-window throughput probe, and to store, in persistent state, both large window and standard performance measurements.

To make experiment results available to Forecasters, Sensors contact Persis-

tent State processes to store the information. The location of the Persistent State process that a Sensor will use for each of the measurements it gathers is specified when the Sensor is configured. When it is initialized, each Sensor registers the location of the Persistent State process that stores its measurement data with the Name Service so that measurement data may be located by name.

5 Forecasting

To generate a forecast, a Forecaster process requests the relevant measurement history from a Persistent State process. Recall that persistent state is stored as a circular queue by Persistent State processes. If the state is being continually updated by a Sensor, the most recent data will be present when a Forecaster makes its request. Ordered by time stamp, the measurements may then be treated as a time series for the purposes of forecasting. The current NWS Forecaster uses such time series of performance measurements to generate forecasts of future measurement values.

An NWS Forecaster works only with time stamp-measurement pairs, and does not currently incorporate any modeling information that is specific to a particular series. Instead, it applies a set of forecasting models to the entire series and dynamically chooses the forecasting technique that has been most accurate over the recent set of measurements. Notice that it is possible to use a forecasting model to "predict" a measurement based on the measurements that come before it in the series. When a forecast of a future value is required, the Forecaster makes predictions for each of the existing measurements in the series. Every forecasting model generates a prediction for each measurement, and a cumulative error measure is tabulated for each model. The model generating the lowest prediction error for the known measurements is then used to make a forecast of future measurement values. This method of dynamically identifying a forecasting model has been shown to yield forecasts that are equivalent to, or slightly better than, the best forecasting model in the set [30].

The advantage of this adaptive approach is that it is ultimately non-parametric and, as such, can be applied to any time series presented to the Forecaster. While the individual forecasting methods themselves may require specific parameters, we can include different fixed parameterizations of a particular method with the assurance that the most accurate parameterization will be chosen.

To allow new forecasting techniques to be integrated easily, the Forecaster process consists of a driver and a set of compile-time determined prediction

modules. The prediction module interface is well-defined, and each module is assumed to implement a different forecasting model. When a forecast is required for a particular time series, the driver presents the time series to each prediction module via the interface, and a forecast for the next value must be returned. The driver keeps track of which prediction module yields the lowest aggregate error measure over time and reports the forecast returned by that module. Any method that can be coded in C which accepts a time series and produces a short-term forecast can be integrated with the system.

5.1 Example Forecasting Results

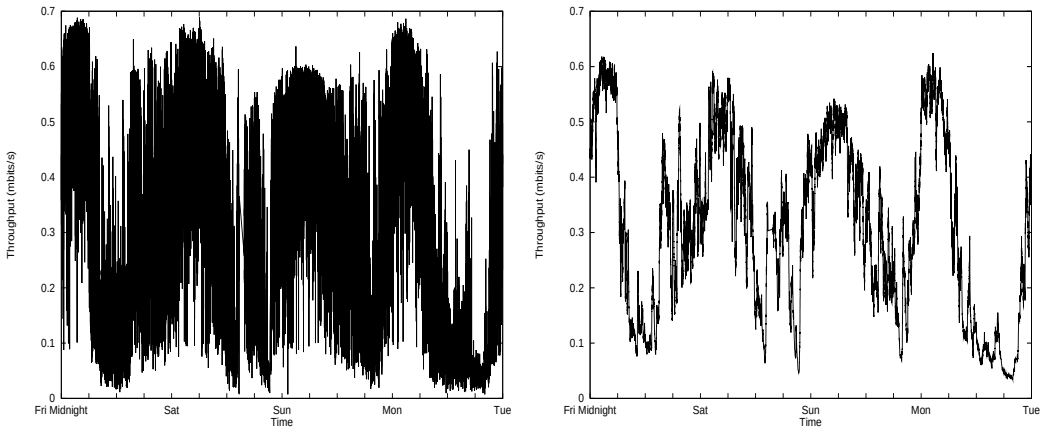

Fig. 3. The left graph shows four days of bandwidth measurements between UC Santa Barbara and Kansas State University. The right graph shows the corresponding NWS forecast values.

Figure 3 shows four-day traces of bandwidth measurements and forecasts generated by the NWS. Long-term trends in the measurements can be seen – throughput tends to peak in the early morning hours, then drop sharply in the afternoon and evening hours before picking up again as midnight approaches. However, the large amount of local variance in the data obscures these trends and limits the usefulness of performance predictions based solely on current measurements. The analysis used by the NWS forecaster allows it to filter local noise and accurately track ongoing long-term trends.

5.2 Incorporating Additional Forecasting Techniques

We wished to be able to extend the NWS to incorporate other forecasting services as they become available. In particular, more parametric modeling approaches, each with their own set of abstractions, are certain to yield good forecasts for specific resources. We would like to be able to incorporate other statistical forecasting methodologies such as Semi-Nonparametric Time Series

Analysis (SNP) [14], automated Box-Jenkins [4] techniques, and wavelet-based models [25]. Since these more sophisticated techniques have larger computational requirements, generating forecasts on-demand according to the requirements of the prediction module interface will not be feasible. We have designed the NWS so that new complete Forecaster processes may be incorporated within the system. Indeed, any process that can retrieve performance data from persistent state (via the Name Service and Persistent State processes) and register itself with the Name Server as a Forecaster can be added dynamically, while the system is running.

6 Reporting Interface

The NWS exports a lightweight and portable C API that contacts the system via sockets so that applications can quickly retrieve short term performance forecasts. For infrequent or casual users, the system also provides continuous access to NWS forecasts through the world wide web.

6.1 C API

The programming interface provided to applications is intended to be lightweight and easily integrated into applications written for systems such as Legion [18], Globus [12], Condor [27], MPI [11], and PVM [17]. Two functions make up this lightweight interface and separate the two phases of a forecaster connection, `InitForecaster()` and `RequestForecasts()`.

The `InitForecaster()` function opens a socket connection to a Forecaster and passes a list of requested forecasts. This Forecaster spawns a new forecaster process to handle the client request. This child forecaster contacts the Persistent State processes that maintain performance measurement data for the requested forecasts, retrieves a recent history of measurements and initiates the forecasting sequence by invoking an initialization function for each predictor configured into the Forecaster. Priming forecasts in this manner allows each predictor to perform initialization early so that subsequent forecasts may be delivered more quickly.

When the application is ready to retrieve forecast data, it calls `RequestForecasts()` to send a request message over the previously-established connection. After updating each predictor with any measurements that have been generated since the call to `InitForecaster()`, the connected forecaster process returns a list of forecasts to the application. Because the forecaster process retrieves only newly-generated data from persistent storage when a request is received,

the time required to generate a forecast can be controlled by the application. An application that requests forecasts frequently will experience a shorter response time than one that waits for long periods between requests. Our experience shows that applications that make frequent requests can receive forecasts in near real time. The forecaster remains available to provide additional forecasts until the application exits.

6.2 CGI Interface

Interactive access to Forecasters is provided by a set of CGI programs [23]. These programs generate time series graphs of performance measurements and forecasts. Trends recognized and followed by the forecasting system are easily discovered when shown in time series form. To allow users to search for either long-term or short-term trends, the web interface provides ways to display graphs showing information taken over time periods of various lengths. Other options allow users to specify the image format and resolution and whether or not the graph should be continuously updated as the NWS system generates additional measurements and forecasts.

7 Sensor Control

To make the system long-lived despite the lossy network connections and intermittent machine failures that occur in any large distributed setting, the NWS relies on adaptive and replicated control strategies. In particular, the Sensors use adaptive time-out discovery and a distributed leader election protocol [15] to remain stable while, at the same time, limiting the load they introduce.

The NWS attempts to measure end-to-end network performance between all possible network Sensor pairs. However, all-to-all network Sensor communication would consume a considerable amount of resources (both on the individual host machines and on the interconnection network) if it were run asynchronously using the entire network Sensor population. The possibility that Sensor probes would collide and thereby measure the effect of Sensor traffic increases quadratically with the number of Sensors. To avoid Sensor contention and to provide a scalable way to generate all-to-all network performance measurements, the network Sensors are organized as a hierarchy of Sensor sets called **cliques**. Each Sensor participating in a clique conducts inter-machine experiments with every other clique member, but not with Sensors outside the clique.

Sensors can participate in multiple cliques simultaneously, so the Sensor pop-

ulation may be organized into a hierarchy by defining different cliques for each level in the hierarchy and promoting one representative Sensor from each clique to also participate in the clique at the next higher level.

For example, consider a Sensor population consisting of Sensor processes running on 5 workstations in the UCSD Parallel Computation Laboratory (PCL), 5 workstations at the San Diego Supercomputer Center (SDSC) [5] , and 5 workstations at the University of Tennessee (UTenn) (Figure 4). The most accurate way to measure the end-to-end performance between all 15 Sensors is to periodically conduct the $15^2 - 15 = 210$ network probes required to match all possible Sensor pairs. However, the performance of a network connection between the PCL and SDSC is dominated by that of a UCSD campus-wide ATM backbone link which must be traversed *en route*. All processes running in the PCL observe approximately the same network performance when communicating with all processes at SDSC. Operating system-specific scheduling vagrancies, buffer management strategies, etc. make this assertion an approximation. With less accuracy, then, it is enough to probe a single PCL-SDSC process pair to determine what the performance of any PCL-SDSC connection will be. Similarly, between the SDSC and UTenn, the performance will be governed by that of the general Internet. Indeed, the PCL and SDSC share all but the campus backbone hop. As the Internet performance dominates, an SDSC-UTenn measurement can represent any PCL-UTenn communication. To organize this Sensor population as an efficient hierarchy of cliques, we define a PCL clique, containing the PCL Sensors, an SDSC clique containing SDSC Sensors, and a UTenn clique containing UTenn sensors. We then define a UCSD-Campus clique in which one of the PCL machines and one of the SDSC Sensors also participate. At the top level, we define a National clique in which one of the SDSC Sensors and one of the UTenn Sensors participate. The end-to-end performance of an arbitrary pair of Sensors, then, is represented by the end-to-end performance between Sensors in the nearest common ancestral clique in the hierarchy.

We may choose (and we can reconfigure dynamically) the PCL to have subcliques of its own. Similarly, if a new site wishes to join the National clique, its representative Sensor can be added dynamically. Further, since the cliques are independent, it is possible to impose different "virtual" hierarchies over the same Sensor population.

To reduce contention within a clique, only a single clique member conducts experiments at any given time. This policy is implemented by passing a clique token among member Sensors. The token contains an ordered list of all Sensors in the clique, which is used to implement token recovery (described below). Holding the token gives a Sensor the "right" to conduct any and all network

[5] SDSC is located on the UCSD campus.

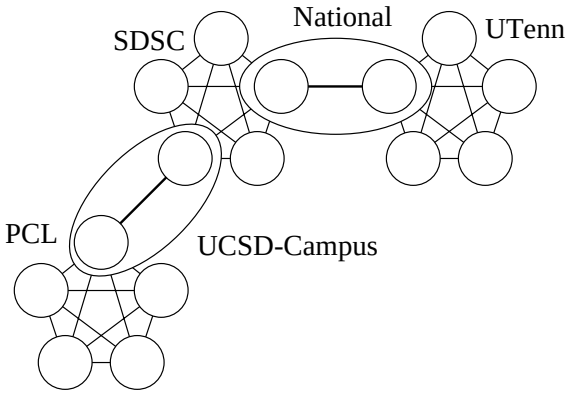

Fig. 4. Example Clique Organization

probes that involve other Sensors in the clique. When it has exhausted the list of probes it wishes to conduct, it passes the token to the next clique member. Once the token has visited all Sensors in the clique, it is returned to the initiating Sensor (the **leader**) which is then responsible for re-initiating it. The periodicity with which the clique leader re-initiates the token controls the periodicity with which each Sensor conducts its probes.

Within a clique, the token may be unavailable either because a Sensor holding it has failed, or because the network has partitioned, isolating one or more Sensors from the one holding the token. Any token recovery mechanism must be able to ensure that the system continues to function under either circumstance. Before the leader re-initiates a token it times a complete token circuit and sets a time-out value for the token (we describe the method by which it determines this time-out value in the next subsection). Once the time-out has been determined, it is carried in the token when it is re-initiated. Each Sensor then calculates a local time-out based on the last time it held the token and the time-out that the leader has determined. If the local time-out expires before a Sensor receives the token again it assumes that either the token has been lost or the network has partitioned. It then generates a new token, marks itself as its leader, and initiates it into the system.

In this way, if the token has been lost due a Sensor failure, a new one will be initiated. Note that the Sensors within a clique receive the token in a particular order, and, therefore, time-out in the same order. So the expected behavior is that the next Sensor in the list will become the leader by initiating the new token. Alternatively, if the network partitions into disjoint sets, at least one token will be started in each set when the time-out occurs. It is possible, using this protocol, for multiple tokens to be present if a time-out occurs but the network has not partitioned, or if a true partition has been eliminated and the partition sets merged. To prevent multiple tokens from consuming network resources indefinitely, tokens are sequenced, and any Sensor encountering an old token discards it rather than propogating it to the next Sensor.

7.1 Adaptive Time-out Discovery

The stability of the token protocol depends on the clique leader's ability to determine when the token should be timed out. If the time-out value is too small, extra tokens will be spawned, consuming greater amounts of network resource and increasing the intrusiveness of the system. If the time-out value is too large, the system may remain quiescent while it waits for some Sensor to time-out and re-initiate the token after a failure. Moreover, the token circuit time is affected by any performance variations in the network the Sensors are using to communicate with each other. The Sensors, therefore, require a prediction of what the time-out value should be, given the performance variations of the network. To make this prediction, the Sensors use the prediction techniques that are integrated with the Forecasters. The clique leader passes a time series of circuit times to a local Forecaster interface and receives back a predicted circuit time and an estimate of the variance associated with that prediction. The time-out is then set to be the estimate plus three estimated standard deviations. When a token times out, the time-out is increased by a fixed amount until the system can "relearn" what the new time-out value should be. In this way, each clique adaptively discovers what the appropriate time-out value should be, given the dynamically changing performance characteristics of the underlying system.

8 Related Work

Resource performance monitoring and forecasting is an active area of research. Internet performance monitoring and analysis tools such as `TReno` [22], `Pathchar` [19], and Carter and Crovella's `bprobe/cprobe` [5,6] attempt to discern Internet congestion characteristics by actively probing the network between designated hosts. We have attempted to design the NWS Sensor interface so that data from these tools can easily be incorporated for forecasting. Topology-d [24] is similar to the NWS in that it conducts a series of performance experiments (using both UDP/IP and TCP/IP) and then automatically analyzes the resulting data. One of its goals is to provide resource scheduling mechanisms such as *Smart Clients* [33], AppLeS [3], and MARS [16] with information depicting the "state" of the network. Important differences, however, concern Topology-d's scalability and periodicity. The performance topology graph it produces is calculated relatively infrequently (once per hour in [24]) using N^2 measurements. The NWS is attempting to capture and forecast higher-frequency dynamics. Typically, NWS network Sensors make measurements once every 10 to 60 seconds. The clique protocol and clique hierarchy allow measurements to be taken at this frequency with limited intrusiveness while also providing scalability. Also, the NWS measures and forecasts the per-

formance of resources other than the network. ReMoS [21] is a generalizable resource monitoring system for network applications. It maintains both static and dynamically changing information, but it does not, at present, include a forecasting component [9]. Its API for accessing the information, however, is similar to that provided by the Globus MDS [10] and Legion Resource Directory [8], but more focused on network information. It should be possible to integrate NWS forecasting techniques with both Topology-d and ReMoS as the relevant APIs are simple and portable.

9 Conclusions and Future Work

The implementation of the NWS relies on adaptivity to enable stability, accuracy, non-intrusiveness, and extensibility. For example, an early version of this implementation used fixed time-outs to control clique-token recovery. We found that such a fixed time-out tended to cause cliques either to pause for long periods of time or to initiate the clique recovery algorithm frequently. Even local-area cliques experienced enough variation to make fixed time-outs impractical. Moreover, using periodic local clocks (i.e. each Sensor probes the network according to its own local periodicity) causes Sensor contention that is statistically significant [31]. Our implementation of the NWS, therefore, gathers more accurate information as a result of its adaptive behavior.

The adaptive forecasting model selection algorithm discussed in Section 5 and [30,29,31] allows the Forecasters to operate in a non-parametric way which promotes extensibility in two ways. First, new and different performance measurement time series may be considered easily. Any series may be presented and, assuming that the suite of models is rich enough, a forecast can be obtained. In [31] we showed a comparison of the forecasting accuracy between sophisticated time series models based on maximum likelihood techniques [14] and the forecasting suite we have implemented (described in [30]). The performance of the currently implemented suite is excellent compared to more powerful techniques for a variety of different metacomputing performance measurements. Second, the adaptive method allows new forecasting models to be incorporated easily. Any model that can be implemented using the predictor module interface can be added to the driver loop in the Forecaster. If it is successful (in terms of its forecasting error performance) it will be chosen.

The longevity of the system and its potential ubiquity stem from its stability, the robustness of its implementation, and its scalability. The implementation platform of TCP/IP sockets and Unix provides a robust and portable set of programming abstractions for a large variety of metacomputing settings. In particular, TCP is well suited to both local area and wide area network settings. The clique abstraction implemented by NWS Sensors provides for

scalability and stability in the system along with limiting its intrusiveness. Perhaps most important is the flexibility that the cliques support. It is possible, for example, to build star topologies [24] or other virtual topologies by defining different sets of overlapping cliques. We plan to use the NWS as a vehicle for our future research in performance monitoring and forecasting.

We plan to continue to enhance the NWS both by adopting new metacomputing standards as they become available, and by incorporating the fruits of the research that is facilitated by the system itself. We are currently working to implement the Name Server using Lightweight Directory Access Protocol (LDAP) [32] as this facility is becoming more commonly available. In addition, we are exploring new forecasting methodologies and new performance monitoring facilities appropriate for different distributed computing environments (e.g. Java).

References

[1] D. Andresen and T. McCune. Towards a hierarchical scheduling system for distributed www server clusters. In *Proc. of the Seventh IEEE International Symp osium on High Performance Distributed Computing(HPDC7) (to appear)*, Chicago, Illinois, July 1998. IEEE Computer Society.

[2] F. Berman. *Computational Grids: The Future of High-Performance Computing, C. Kesselman, and I. Foster, editors.* to appear, 1998.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.

[4] G. Box, G. Jenkins, and G. Reinsel. *Time Series Analysis, Forecasting, and Control, 3rd edition.* Prentice Hall, 1994.

[5] R. Carter and M. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report TR-96-007, Boston University, 1996. available from `http://cs-www.bu.edu/students/grads/carter/papers.html`.

[6] R. Carter and M. Crovella. Measuring bottleneck link speed in packet-switched networks. Technical Report TR-96-006, Boston University, 1996. available from `http://cs-www.bu.edu/students/grads/carter/papers.html`.

[7] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *Proc. of Supercomputing'96, Pittsburgh.* Department of Computer Science, University of Tennessee, Knoxville, 1996.

[8] S. J. Chapin, J. Karpovich, and A. Grimshaw. Resource management in legion. Technical Report CS-98-09, University of Virginia, Department of Computer Science, May 1998.

[9] T. DeWitt, B. Lowecamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource monitoring system for network-aware applications. Technical Report

CMU-CS-97-194, Carnegie-Mellon University, december 1997. available from `http://www.cs.cmu.edu/afs/cs/user/jass/www/index.html`.

[10] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.

[11] M. P. I. Forum. Mpi: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, 1994.

[12] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997. to appear.

[13] I. Foster and C. Kesselman. The globus project: A status report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[14] R. Gallant and G. Tauchen. Snp: A program for nonparametric time series analysis. In `http://www.econ.duke.edu/Papers/Abstracts/abstract.95.26.html`.

[15] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):49–59, Jan 1982.

[16] J. Gehrinf and A. Reinfeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. *Proceedings of Future general Computer Systems*, 1996.

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

[18] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The next logical step towrd a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 1994.

[19] V. Jacobson. A tool to infer characteristics of internet paths. available from `ftp://ftp.ee.lbl.gov/pathchar`.

[20] R. Jones. `http://www.cup.hp.com/netperf/netperfpage.html`. Netperf: a network performance monitoring tool.

[21] B. Lowecamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, August 1998. available from `http://www.cs.cmu.edu/afs/jass/www/papers.html`.

[22] M. Mathis and J. Madhavi. Diagnosing internet congetstion with a transport layer performance tool. In *Proceedings of INET '96*, 1996.

[23] Network weather service. `http://nws.npaci.edu/`.

[24] K. Obraczka and G. Gheorghiu. The performance of a service for network-aware applications. In *Proceedings of 2nd SIGMETRICS Conference on Parallel and Distributed Tools*, August 1998. to appear.

[25] R. Ogden. *Essential Wavelets for Statistical Applications and Data Analysis*. Birkhauser, 1997.

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

[27] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[28] vBNS. `http://www.vbns.net`.

[29] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997. to appear.

[30] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. also available from http://www.cs.ucsd.edu/users/rich/publications.html.

[31] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of Supercomputing 1997*, November 1997.

[32] W. Yeong, T. Howes, and S. Kille". Lightweight directory access protocol, March 1995. RFC 1777.

[33] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build sclable services. In *Proceedings of the USENIX 1997 Technical Conference*, 1997.