JavaNws: The Network Weather Service for the Desktop

Chandra Krintz
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
ckrintz@cs.ucsd.edu

Rich Wolski*
Computer Science Department
University of Tennessee, Knoxville
Knoxville, TN 37996
rich@cs.utk.edu

ABSTRACT

As research and implementation continue to facilitate high performance computing in Java, applications can benefit from resource management and prediction tools. In this work, we present such a tool for network round trip time and bandwidth between a user's desktop and any machine running a web server [1]. JavaNws is a Java implementation and extension of a powerful subset of the Network Weather Service (NWS), a performance prediction toolkit that dynamically characterizes and forecasts the performance available to an application. The JavaNws capitalizes on the Java execution model to eliminate the need for NWS installation and login accounts on the machines of interest. In addition, we provide a quantitative equivalence study of the Java and C socket interface and show that the data collected by the JavaNws is as predictable as, that collected by the NWS (using C).

1. INTRODUCTION

The Internet today provides access to distributed resources throughout the world. The network performance available to an arbitrary user program in this environment, however, is highly variable. Available network performance (latency and bandwidth) fluctuates over short time-scales making it difficult for a user to make informed decisions about whether or not it is practical to use the network (for distributed execution or download) at any given time. Users have access to a variety of network performance monitoring tools (SNMP [4], netperf [13], pathchar [11], the Unix `ping` command, etc. as well as a raft of others listed with [9] and [10]) but all of these tools provide an estimate of performance conditions *that have already occurred*. A user wishing to decide between two equivalent download sites must assume that the conditions that have been observed will persist until the download is complete. That is, the user uses the current conditions as a *prediction* of what the conditions will be a short time into the future. However, statistical analysis of network performance data indicates that the last value observed is rarely the best predictor of future network performance [20]. Furthermore, many of the available tools for determining current conditions do not use the same network protocols as user applications, nor do they measure network utilization in terms of aggregate packet traffic. To make an effective determination, users require application level predictions of available network performance.

To solve these problems, we have developed the Network Weather Service [21, 20, 17] (NWS) at the University of Tennessee, Knoxville. The NWS is a distributed service that provides users with measurements of current network performance and accurate predictions of short-term future performance deliverable to an application or download. The NWS components operate with no special privileges and use TCP/IP — a protocol commonly used in user applications and browser downloads – to measure network availability. The measurements are treated as time series and a set of adaptive statistical forecasting models are applied to each to make short-term predictions of available network performance.

The NWS, however, requires hard collaboration between processes. A user must install the package on his or her own machine and understand the interfaces in order to make queries to the NWS subsystems. At the very least, the user must have login accounts on any machine of interest on which NWS is not yet installed. Lastly, a user may only be interested in the network resource; the NWS measures and forecasts many resources, e.g., cpu, memory, etc., in addition to the network.

In this paper, we describe an implementation of NWS functionality using Java [14, 6] and applets to circumvent the need for the user to explicitly install and maintain an NWS network monitoring process. This tool, the *JavaNws*, enables an arbitrary user to simply click on a link and visualize the current and future, predicted performance of the network between the desktop and the web server.

JavaNws is the first to tool to visualize NWS data dynam-

*Supported by NSF grant ASC-9701333 and the NSF National Partnership for Advanced Computational Infrastructure

[1] This assumes that the user's browser is capable of supporting Java 1.1 and above

ically and continuously. JavaNws provides a framework for user customization of resource experiments dynamically. Parameters including adjustment in periodicity, and size of packets used in bandwidth tests, can be modified to match those of interest by the user. In addition, no installation or user accounts are necessary to use the robust network prediction services of the NWS and to provide short-term forecasts of future network performance.

The tool is completely operational and installed at the NSF computational centers: the National Partnership for Advanced Computational Infrastructure (NPACI) and the National Center for Supercomputing Applications (NCSA). Each "center" offers access to machines located at various sites throughout the U.S. Using JavaNws, users can choose the site that *will* offer the best network connectivity to his or her desktop. JavaNws has been incorporated into the NPACI HotPage [16] – web based access facility available to NPACI users. In addition, the JavaNws was demonstrated by NPACI at SC99 in Portland, Oregon where it was used characterize the performance between machines located at the conference and different partner sites.

We selected Java [14, 6] as our implementation language for its applet execution model as well as its wide spread availability and acceptance as an Internet computing language. The Java execution model allows a program to be downloaded to a user desktop and executed locally as opposed to remotely. This model enables us to establish an interactive session between the applet and the remote source machine in which we can measure the current network performance. In addition, as research and implementation continue to facilitate high performance computing in Java, performance-oriented Java applications will benefit from resource measurement and prediction tools [18, 15, 2, 12].

This project, however, is not purely an engineering endeavor; this study lends insight into the overhead associated with using Java TCP sockets. We discuss the differences between the Java the C language measurements and provide a quantitative comparison between TCP socket implementation in the two languages. In addition, we show that the accuracy with which network performance can be predicted is the same for C and Java using TCP/IP and sockets.

In the next section we detail the implementation of the JavaNws. Section 3 describes the experimental methodology we used in this study. In Section 4.2, we analyze Java and C performance results based on long-running performance traces. Because statistical comparison can be difficult for non-Normal data, we also analyze the predictability of each methodology in Section 5. In the final sections we detail the our future directions and conclude (Sections 6 and 7 respectively).

2. IMPLEMENTATION OF JAVANWS

The JavaNws is a Java implementation of the NWS network measurement and forecasting subsystems (see [21] and [20] for a complete description of NWS functionality and forecasting techniques). The JavaNws provides a graphical display of the performance data to allow users to visualize the network performance (actual and predicted) between the server and their desktop in real-time. For example, a user

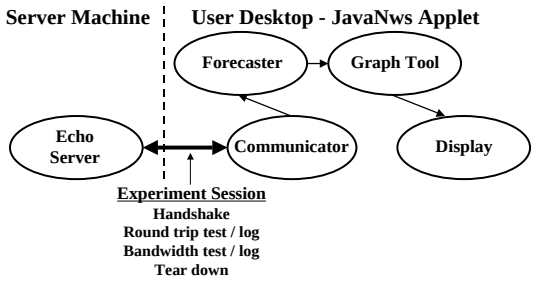

Figure 1: The JavaNws Architectural Design

may decide to download a piece of software from an arbitrary site on the World Wide Web (WWW). In order to determine if the download time is feasible, he or she can first click on a JavaNws link provided by the site to view the current as well as future, predicted network conditions. In addition, the download site can provide links to JavaNws at its mirrored sites allowing the user to use the predicted network performance to make informed decisions about the download times from each site. Previous work with the NWS and Java-based applications indicates that basing transfer decisions on NWS forecast data can dramatically improve execution performance [5, 19].

The JavaNws consists of two parts: The applet that executes on the user's desktop and the server program (called the Echo Server) located at the machine from which the applet is downloaded. When a user clicks on a JavaNws link, a Common Gateway Interface (CGI) program is executed that invokes the Echo Server in the background and then initiates transfer of the applet to the user's desktop for execution.

2.1 The JavaNws Applet

The majority of the JavaNws functionality is in the applet. This applet is a Java program that consists of a communicator, a forecaster, a graph tool, and a graphical user interface (GUI) as depicted in Figure 1. The Communicator and the Forecaster are direct translations of the NWS C-versions. The graph tool is an extended version of a freely available graph display tool called XYGraph[1], and the display was designed and implemented from scratch to enable visualization of the graphs and to allow dynamic customization of experiment parameters by the user.

The applet communicates with the server machine from which it was downloaded. To measure network performance, the Communicator conducts in a series of communication *probes* between itself and the CGI-invoked Echo Server running on the server machine. During each probe, measurements are taken of round trip time and bandwidth. These experimental results are passed to the Forecaster so that predictions can be made for the network performance at the next time step. The predictions and the measurement data are then passed to the graph tool which updates the graph with the new values and incorporates them in the interactive display. We briefly discuss the implementation of this design in the following sections.

2.1.1 The Communicator

The Communicator performs a series of experiments to measure the round trip time and available bandwidth between

the desktop and the server machine. The Communicator (client) and the Echo Server first handshake to establish an experiment session. A protocol is then used to measure round trip time and bandwidth. The protocol consists of a series of packet exchanges that are timed by the Communicator.

For round trip time, a two-byte packet is exchanged. To determine the extent to which the Nagle algorithm [2] effects the transfer time, two tests are performed; one with the Nagle effect off and one with it on (in Java: setTcpNoDelay(false)). The Nagle algorithm is used by TCP to improve network performance when many small packets are being sent. With the Nagle algorithm, TCP waits to send many small packets at once; if no other packets are sent, TCP eventually forwards the small packet. We report the round trip time with the Nagle effect to the user. The non-Nagle-impeded round-trip time, however, is used to adjust subsequent bandwidth measurements (discussed below).

Following the round trip time experiments, the Communicator reports the measurement to the Echo Server. Next, a probe is conducted to determine bandwidth. The applet times a "long" transfer (64KB by default, although the tool allows the user to set the probe duration) and calculates the resulting bandwidth. This timed exchange consists of a large packet from the Communicator to the Echo Server and a 2 byte packet from the Echo Server to the Communicator acknowledging the completion of the probe. Notice that the complete exchange includes round-trip time as the Communicator must wait for the acknowledgement before finishing its timing. If the user chooses a small probe size and the round-trip time is large, this extra time can be significant. To account for this, we subtract the *predicted* round-trip time calculated from the non-Nagle exchange that occurs previously from the time observed for a bandwidth probe. That is, we predict what the round-trip time will be for the bandwidth probe period and then subtract that prediction from the actual probe time to account for the extra acknowledgement time.

In this paper, we report data with and without the Nagle effect to show the differences between the Java version and the C, respectively. We do not reduce the bandwidth time by the round-trip time prediction, however, as we did not want to cloud the comparison between observed C and Java performance with questions of forecast accuracy. Rather, we analyze the performance of each strictly in terms of the observed probe times.

2.1.2 The Forecaster

The Forecaster is a Java implementation of the NWS forecasters. A detailed description of the the NWS forecasters can be found in [20]. In short, the Forecaster consists of a set of independent forecasting algorithms [7, 3, 8], each of which produces a one-step-ahead forecast from a given time series. At each time step, the measurement data is received from the Communicator and compared to the forecast produced by each forecasting algorithm for that time step. The difference between each forecast and the measurement it is forecasting is the *forecast error*. The Forecaster records the the mean square forecasting error (MSE) associated with each forecasting algorithm. To make a single prediction for the next time step, the algorithm having the smallest MSE is chosen. It is this most-accurate-up-until-now forecast that is reported to the user as the NWS prediction. Note that the Forecaster keeps a separate time series for round-trip time, Nagle-impeded round-trip-time, and bandwidth.

2.1.3 The Display

To display the measurements and predictions in a graphical format, we incorporate a freely available graph tool called XYGraph [1]. The tool takes a file of data and graphs it in various formats. We extended the tool to enable graphing of a history of measurements and predictions. In order for the display to appear continuous, the graph is redrawn as each experiment is performed; the data values appear to move across the graph. The forecasts are displayed one time step ahead of the actual measurements since the values indicate the performance deliverable at the next time step. The graphs are displayed on frames separate from the console. Each resource has its own set of graphs generated by the graph tool. These frames also contain information about the experiments: the client and server host names, the actual measurements, the forecasted measurements and the time of each. The user can also choose to view measurement data for given resource on the same graph as the forecast data or separately.

Figure 2 shows each resource graph (for round trip time and bandwidth) during execution with both the actual and forecasted measurements displayed on the same graph. The Y-axis is the round trip time in milliseconds and the bandwidth in Mb/sec, for the top and bottom graphs, respectively. The X-axis is the time at which the measurement was taken; experiments were performed at approximately 12 second intervals. The dark points are the forecast data and the light points are the actual measurements. At a time step at which only one point appears, the forecast and the actual measurement are the same.

The final piece of the JavaNws is a GUI that combines all of the components together. The interface consists of an interactive console and graph displays. The console allows the user to modify the parameters of the experiment. The user can use the console to view or hide the graphs; even with the graphs hidden, however, the summarized data is displayed on the console and experiments continue to run. The information provided on the console includes current time, resource measurements (round trip time and bandwidth) from last experiment, the next predicted value for each resource, as well as the accuracy (measured in cumulative mean absolute error) of the predictions made so far. The user can modify the time between measurements so that they occur every 1 second to 5 minutes. In addition, the size of the packet in the bandwidth experiment can be 64KB, 128KB, 256KB, or 512KB. Finally, the user can use the console to suspend and resume the experiments at any time. Figure 3 shows the console during execution.

[2] We will refer to the combination of the Nagle small-packet-avoidance algorithm and the delayed acknowledgement algorithm (which avoids silly-window syndrome) together as the "Nagle algorithm" throughout this paper. Although, strictly speaking they are separate optimizations, they both manifest themselves as potential delays in the end-to-end observed performance.

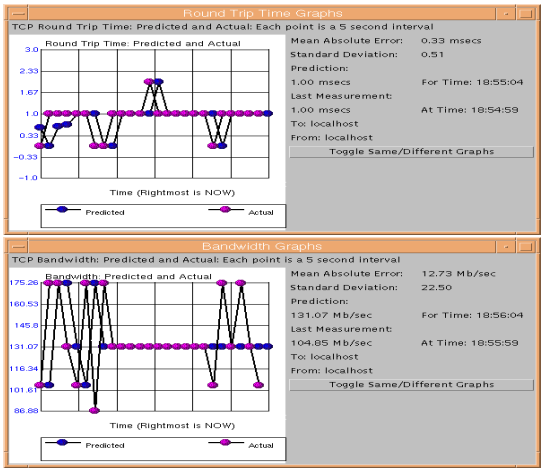

Figure 2: A view of the round trip time (TOP) and bandwidth (BOTTOM) graphs in action. Dark points are predicted values and light points are the actual measurements. The right-most point in each graph is the value predicted to occur in the next (future) time step. Where there appears to be only a single point at a given time step, the predicted value is the same as the actual measurement (it was correctly predicted).

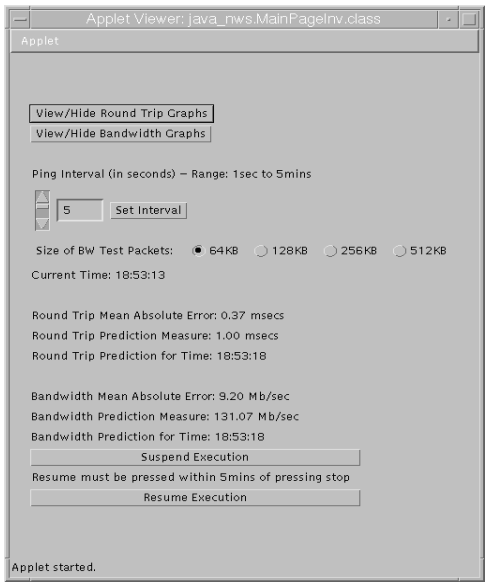

Figure 3: A view of the console in action.

Table 1: Locations of Machines Used in Experiments.

a-g	ash (San Francisco) to gibson (UTK) via the Internet, Java v1.1.3
c-d	conundrum (UCSD) to dsi (UTK) via the vBNS, Java v1.1.3
f-n	fender (UTK) to ncni (UNC) via the vBNS, Java v1.1.7
p-k	pacers (UTK) to kongo (UCSD) via the vBNS, Java v1.1.7

Table 2: Types of Machines Used in Experiments.

ash	Sparc Ultra I, Solaris 5.6, 167Mhz processor, 256 memory
conundrum	Sparc Station 5, Solaris 5.6, 110Mhz processor, 64MB memory
dsi	RS6000, AIX 4.3, 332Mhz processor, 200MB memory
ncni	RS6000, AIX 4.3, 332Mhz processor, 200MB memory
fender	x86, Linux, 400Mhz processor, 256MB memory
gibson	x86, Linux, 400Mhz processor, 512MB memory
kongo	Sparc Ultra I, Solaris 5.6, 166Mhz processor, 192MB memory
pacers	x86, Linux, 300Mhz processor, 512MB memory

3. EXPERIMENTAL METHODOLOGY

Table 1 gives the location of machines for each pair of hosts, the predominant network technology separating the two, and the version of Java that we used to generate the results presented in this paper. The locations include San Francisco (MetaExchange.com), the University of California, San Diego (UCSD), the University of Tennessee, Knoxville (UTK), and the University of North Carolina (UNC). UCSD, UNC, and UTK are vBNS sites. The vBNS is an experimental transcontinental ATM-OC3 network sponsored by NSF that can be used for large-scale and wide-area network studies. It is characterized by high-bandwidths and relatively high round-trip times induced by large geographic distance. Our Internet connectivity is common-carrier. The types of machines and operating systems used are in Table 2. In total, we ran experiments between 10 pairs of hosts. We include only 4 pairs of hosts in the results due to space constraints. The 4 data sets we have chosen are representative of all pairs used in the experiments.

All experiments were performed at approximately 12 second intervals over a 24 hour period on weekdays between various machine pairs. Each experiment consisted of measurements first taken by the Java version then by the C version [3]. The Java version is executed using a standard Java interpreter.

The Java version uses a Communicator written in Java communicating with a C program using Unix sockets (called the Echo Server). The C version of the experiments uses a Communicator written in C communicating with the same Echo Server. Each experiment involved an exchange using the

[3]We also performed the measurements in the opposite order (C and then Java) and found similar results

Table 3: Mean and variance values of the C and Java bandwidth data.

	C Version		Java Version	
Host Pair	Mean	Variance	Mean	Variance
a-g	0.61	0.02	0.61	0.02
c-d	0.65	0.05	0.65	0.05
f-n	1.41	0.00	1.66	0.00
p-k	1.06	0.01	1.01	0.01

Table 4: Mean and variance values of the C and Java round-trip time data.

	C Version		Java Version	
Host Pair	Mean	Varience	Mean	Variance
a-g	96	890	96	1007
c-d	94	102	95	2888
f-n	49	3	49	9
p-k	95	175	96	3579

protocol described in Section 2.1.1. The raw data for two pair of hosts (p-k: pacers (UTK) to kongo (UCSD), and c-d: conundrum (UCSD) to dsi (UTK)) is found in Figures 4, 5, and 6. The right graph is the data collected by the C version; the left collected by the Java version. We accumulated similar data for all host pairs.

4. COMPARING RAW JAVA AND C PERFORMANCE

The performance observed during a particular network transfer is a function of the load on the network at the time the transfer is made (as well as underlying network technology). For the networks we used in this study, that load is constantly varying. It is therefore difficult to compare the performance Java and C as it is impossible to test them both under identical load conditions. Even when the C and Java probes are run back-to-back, the network conditions may change between experiments making a pairwise comparison ambiguous. Therefore, we resort to a comparison of statistical characteristics generated from relatively large samples of each, and argue for equivalence (or otherwise) based on these characteristics.

4.1 Comparison based on Moments

Tables 3 and 4 show the sample means and variances for bandwidth and round-trip time (respectively) between four representative host pairs over 24 hours.

Clearly, if each sample is large enough to capture the underlying distributional characteristics of the method it represents, then C and Java are almost equivalent in terms of their first two moments. That is, while individual examples may not be comparable, the mean performance, and the variance in performance between C and Java are equivalent. Unfortunately, since the data is not well approximated by Gaussian distributions, it is difficult to determine the statistical significance of this comparison rigorously. Instead, we observe that in terms of mean (the average behavior) and variance (the average deviation from the mean value) C and Java performance are similar. Often this colloquial level of similarity is within engineering tolerance for many applications.

4.2 Comparison based on Regression Coefficients

Another sample-based technique is to calculate the regression coefficient between samples. We pair values from each sample together based on time stamp (two values taken nearest in time form a pair) and then calculate the linear coefficients based on least-squares regression. Our motivation is to observe how close the derived multiplicative constant is to 1.0 for the different traces. Table 5 shows the results.

The regression coefficient indicates the degree to which C and Java are proportional with respect to their respective means. A value near 1.0 that the best way to predict a Java value from a C value is to multiply the C value by 1.0.

The first column of Table 5 is for bandwidth, the second for round trip time, and the third for round trip time with the Nagle effect. When two sets of measurements come from the same series the least square regression value of their differences is very close to 1.0. This value also provides an estimate of the percent difference between the average measurements (it is the coefficient that the Java value is is multiplied by to get a corresponding C value). If the figure is less than 1.0, then the Java version reported measurements that are less than those reported by C (higher round trip times and lower bandwidth values). For example, the table shows that for the pair a-g (ash-gibson), the Java bandwidth measurements are 1% less on average than the C bandwidth measurements. The round trip time without and with the Nagle effect measurements for this pair of hosts are (on average) 7% and 13% slower, respectively, than that of the C version for this link.

The least squares regression coefficient is impacted by outliers in the data. For bandwidth, there were no obvious outliers. For round-trip time, however, there were occasional values that deviated by two or three orders of magnitude. We do not believe that these values are representative of C or Java effects but rather catastrophic network failures. In order to rectify this problem and report a value unaffected by outliers, we removed outliers for certain experiment sets. The number of values removed were: a-g (2), and f-n (4). The total number of values in each trace is approximately 7200 (every 12 seconds for 24 hours) making these outliers very infrequent. A thorough investigation of the hypothesis that these outliers result from network behavior (and not C or Java performance) is the subject of our future work.

If the least squares regression coefficient is greater than 1.0, then the Java version reported measurements that, on average, were better than that of the C version. Notice that in c-d and f-n the bandwidth measurements from the Java version are better (greater than 1.0) than the C version. This condition also occurred for two other host pairs in data we do not report here due to space constraints. We looked into this anomaly (since we believed that in every case Java should be slower than C due to extra processing overhead of Java socket abstractions and interpretation) and found that the

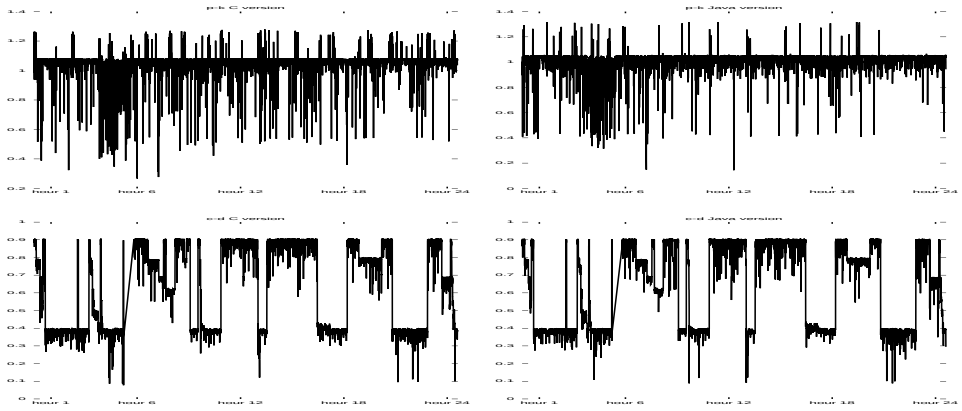

Figure 4: Raw bandwidth data for the pairs of hosts p-k (pacers to kongo) (TOP) and c-d (conundrum to dsi) (BOTTOM). The left graph are the measurements taken by the C version and the right are those taken by the Java version.

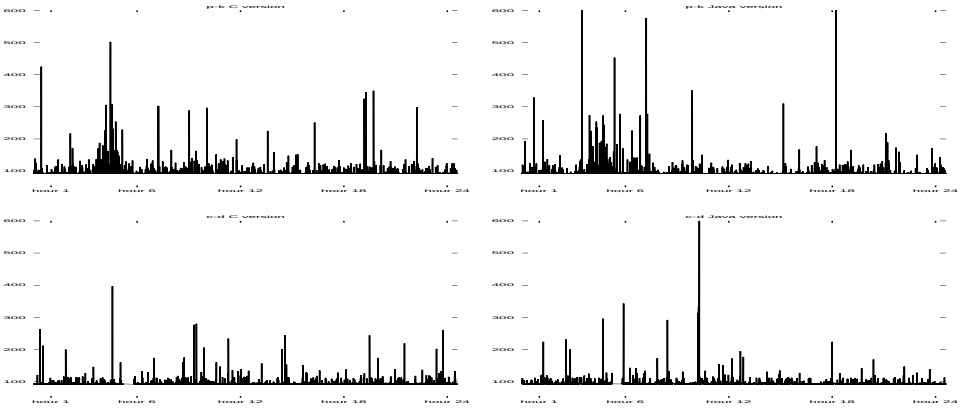

Figure 5: Raw round trip time (without the Nagle effect) data for the host pairs p-k (pacers to kongo) (TOP) and c-d (conundrum to dsi) (BOTTOM). The left graph are the measurements taken by the C version and the right are those taken by the Java version.

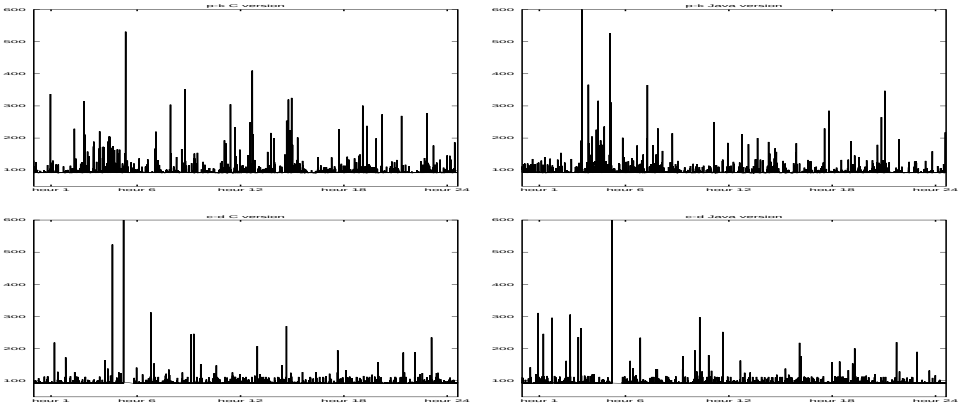

Figure 6: Raw round trip time (with the Nagle effect) data for the host pairs p-k (pacers to kongo) (TOP) and c-d (conundrum to dsi) (BOTTOM). The left graph are the measurements taken by the C version and the right are those taken by the Java version.

Table 5: Least square regression of C to Java measurement values. If the coefficients are equal to 1.0, then the values reported by the Java version are from the same time series as the C version.

Host Pair	Bandwidth	Round Trip Time	Round Trip Time (Nagle)
a-g	0.99	0.93	0.87
c-d	1.01	1.00	0.96
f-n	1.18	1.00	1.00
p-k	0.95	0.99	0.98

operating system on the server end of these three pairs of machines were all IBM RS6000's running operating system (OS) AIX version 4.3. In addition, each receive operation performed by the echo server on these machines occurred in blocks for 512 bytes (the maximum transfer unit (MTU) set as the default during AIX installation). On all other host pairs, the MTU on the server machine ranged from 1KB to 2KB. When we modified the C version to send using a buffer size of 64KB instead of 32KB, the least square bandwidth coefficients changed to 1.00 for all such pairs. This indicates that Java the version we incorporated used a buffers size of 64KB. Only when the receive size is 512 bytes or less, is the overhead of sending 2-32KB buffers (as opposed to 1-64KB buffer) apparent. We would like to control the Java buffer size in order to ensure it is the same as in the C version, but Java version 1.1x does not provide a mechanism for modifying the buffer size. We use the 64KB buffer size C version data for the predictability tests in Section 5. It is a noteworthy point, however, that the buffer size affects predictability. For systems such as Java 1.1x (which uses undocumented, unalterable buffer sizes) the buffer sizes used may interact with the base OS in ways that magnify the uncertainty inherent in the underlying network dynamics.

Alternatively, Java 2 does provide a mechanism for setting the buffer size on sockets. We ran a series of bandwidth experiments on a pair of hosts (k-j) for which Java 2 was available. JavaNws ran on a host located at UCSD (k) and the server was located at UTK (j); and the vBNS was the primary Network between them. The least squares fit between C and Java improved from 0.92 to 1.00 when we used a buffer size of 32KB on both the C and Java versions, again demonstrating the effect of buffer size on observed, end-to-end predictability. The data (Table 6) indicates that controlling the buffer size improves the performance of the Java to equal that of the C version. In the JavaNws implementation we have developed, we use the Java 2 socket buffer interface as available.

Another interesting insight that this data provides us with is that the Nagle algorithm and TCP "delayed acking" strategy seems to effect the Java TCP sockets to a greater extent than C TCP sockets. This effect can be seen by comparing the rightmost column with the middle column of Table 5. When the Nagle algorithm is used, the Java measurements are not as close to the C measurements as when the Nagle algorithm is turned off. We are not, at present, able to discern the nature of this difference although we speculate that

Table 6: Least squares regression of C to Java measurement values using a buffer size of 32KB in both C and Java (using Java 2) versions.

Host Pair	Bandwidth
k-j (Java 1.1.3)	0.92
k-j (Java 1.2.1)	1.00

Java and C are managing their socket buffers differently and the result causes a slight timing difference which the TCP optimizations magnify.

We also learned that the system-supplied data structures used to manage the buffers in the probe programs is significant. In order for the Java version to report measurements linearly similar to those of the C version, we had to ensure that we used byte arrays in Java to write to the socket. This data structure eliminates the overhead of buffering and conversion so that the timings the Java version reported were much closer to those from the C version.

One source of error we were not able to eliminate, however, stems from the difference in clock precision that is available to user-level programs from Java and C. Java returns a rounded long-typed millisecond value when the time is queried and C returns a double value. Small differences between C and Java performance may be caused by this rounding effect.

5. COMPARING JAVA AND C PREDICTABILITY

Table 4 (from the previous Section) indicates that there is a larger variance in the Java version of the round trip time data not seen in the bandwidth data. This difference may be due to the sample size (not enough data) or it may be an indication that probe trace from the Java version is not equivalent to that from the C version. Indeed, recent work indicates that network performance may be heavy-tailed making the problem of determining an appropriate sample size for statistical significance difficult to solve.

Rather than tackling this often intractable problem, we note that most often performance profile data is used to make some form of prediction. If not used for fault diagnosis, performance traces are almost always used to anticipate future performance levels. A user, for example, may examine a recent history of measurements to anticipate the duration a particular network transfer he or she wishes to perform. As such, we frame the problem of determining equivalence in terms of predictability. Note that, again, statistical significance is an issue. As part of our engineering-based approach, however, we report how predictable *in terms of observed prediction error* each series is over suitably long time periods. Regardless of statistical significance, the prediction error is what we observed. The consistency with which we observed it is the basis for our conclusions.

To compare performance trace predictability, we treat each probe trace as a time series and look at the prediction error

generated by the NWS forecasting system (described in Section 2 and more completely in [20]). At each point in the trace, we make a prediction of the succeeding value. The forecasting error is calculated as the difference between a value and the forecast that predicts it.

The advantage of this approach is that it admits the possibility of the underlying distributions changing through time (non-stationarity) since the NWS forecasting techniques are highly adaptive. That is, we do not treat each trace as a sample, but rather as a series, each value of which may depend on the time it is taken.

The following set of graphs compare the predictability of the data collected by the C and Java version of the NWS. Predictability can be compared using a histogram of the error values, where the error value is the difference between the NWS predicted value and the actual value that it predicts. In each case, we form a histogram of 100 bins spanning the difference between the minimum and maximum measured values. Figures 7 and 8 contain the histograms for each pair of hosts for bandwidth (7), and round trip time (8). The left graph is the C version and the right is the Java version. Clearly, the error histograms are almost identical. [4]

Histograms are useful for visualizing distributional data, but prone to variation due to bin size. Indeed, the predictability for the round trip time measurements are difficult to compare using this format, hence we include data for only one pair of hosts (p-k) in Table 8. This difficulty is due to the the presence of infrequent but significant outlying values. Many histogram "buckets" contain very few data elements with one or two buckets containing the large portion of the data. It is difficult to determine visually the contribution that these buckets (that contain very little data each) make to either the difference or similarity between error series.

As an alternative, we present the data in cumulative distribution functional (CDF) form. Figures 9 contain the CDFs of the prediction errors for round trip time for each host pair and version. If overlayed, the C version CDF is nearly identical to the Java version CDF indicating that the two data sets are equally predictable. CDF comparison of round-trip time prediction error for data with the Nagle optimizations enabled also shows near perfect similarity. We eliminate the figure for the sake of brevity.

6. EXTENDING JAVANWS

One limitation of the JavaNws is due to applet execution restrictions; the Java applet may only communicate with the machine from which the applet was loaded (the source machine). This prohibits measurement of the network between the desktop and an arbitrary machine, and between two arbitrary machines. It may be desirable for a user to be able to gather this information from machines on which he has no logins (thereby disallowing installation of the NWS by the user). In addition, the NWS may already be installed on many machines of interest. We are currently extending the JavaNws to access measurements taken by either an extant installation of the NWS (of network performance between arbitrary pairs of machines) or the JavaNws applet (of performance between the server and the desktop) depending upon which is more convenient. The JavaNws forecasters will continue to be used to predict future deliverable performance, regardless of which mechanism provides measurement data.

[4]The histograms are, indeed, so similar that we were not able to superimpose one over the other meaningfully without the use of color.

7. CONCLUSION

We have designed and implemented a Java implementation of the Network Weather Service (NWS). JavaNws is a fully functional tool that allows a user to visualize the current and future, predicted performance of the network between the desktop and any World Wide Web site. The tool capitalizes on the transfer model and availability of Java and is currently being used to facilitate high-performance distributed computing with Java.

A JavaNws applet conducts a series of experiments between the desktop and a server program executing at the site from which the applet was downloaded. JavaNws uses the NWS forecasting algorithms (implemented in Java) to predict the network performance of the near-term. Bandwidth and round trip time (using TCP sockets) measurements and forecasts are then presented to the user in a constantly updated graphical format. The user can customize the parameters dynamically of the experiments (bandwidth test size, test frequency, etc.).

In addition, we provide a quantitative study of the performance differences between Java and C. We use rigorous statistical analysis (mean/variance and least squares regression) to rule out differences between the data sets due to population size, outlying data, and other such anomalies. We show that the data collected by the Java version is equivalent to that by the C version. We also compare the predictability of the JavaNws data to that of the NWS (written in C). We compute error values (differences between predicted and actual values) using the NWS forecasting algorithms to construct histograms and cumulative distribution functions with which to compare the two sets of data (time series). Our detailed analysis establishes that using Java to collect the performance data does not reduce the predictability of the series, i.e., it is as predictable as that of the C version.

8. ACKNOWLEDGEMENTS

Thanks to the referees and others who provided valuable feedback on this work.

9. REFERENCES

[1] G. Bildson. Xygraph. `http://home.sprynet.com/~gbildson/ngraph/xygraph.html`.

[2] B. Blount and S. Chatterjee. An evaluation of java for numerical computing. In *Scientific Programing*, volume 7 (2), pages 97–119, 1999. Special issue on high performance Java compilation and runtime issues.

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

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp), May

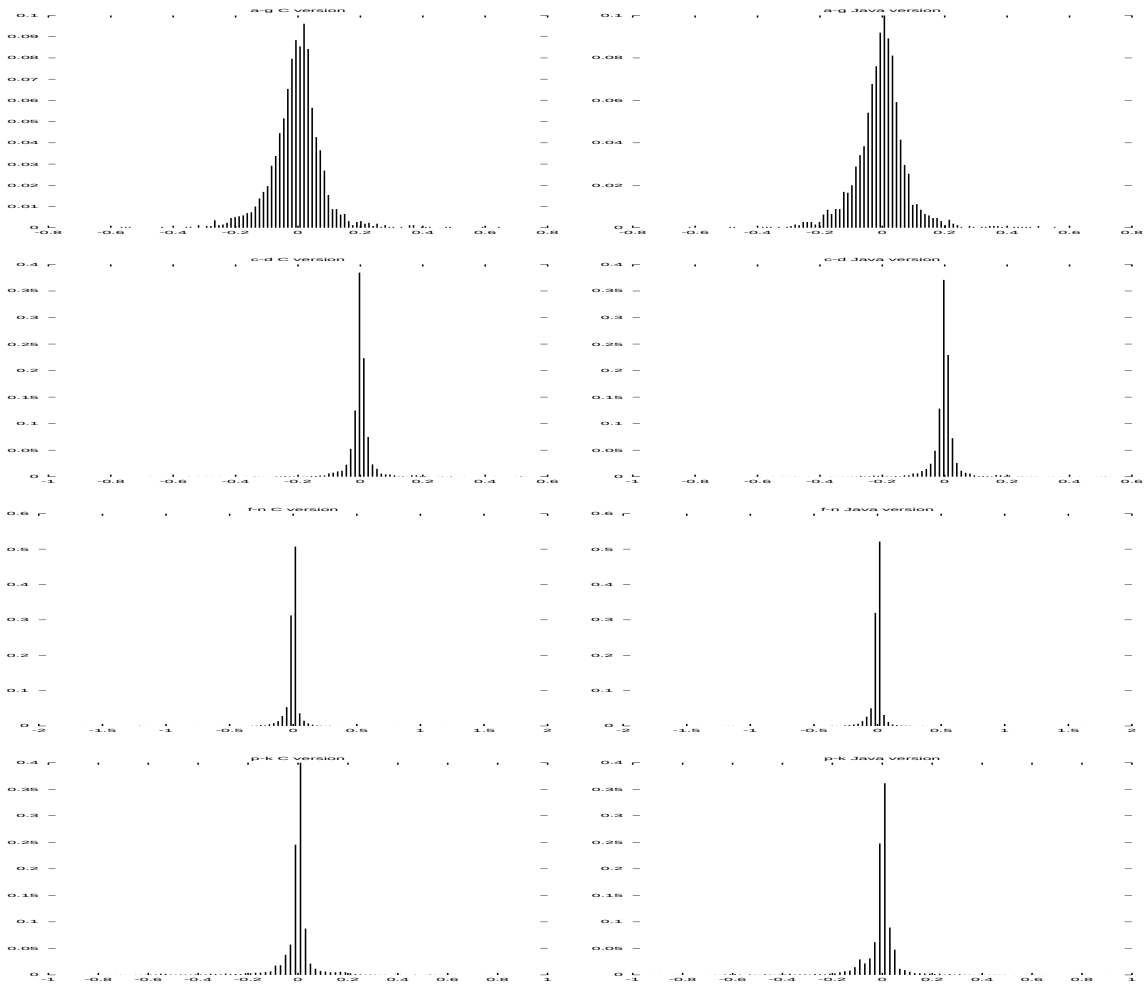

Figure 7: Histograms of error values from bandwidth prediction using measurements from C (left) and Java (right) versions. The graphs indicate that the Java version of the bandwidth data is as predictable as the C version.

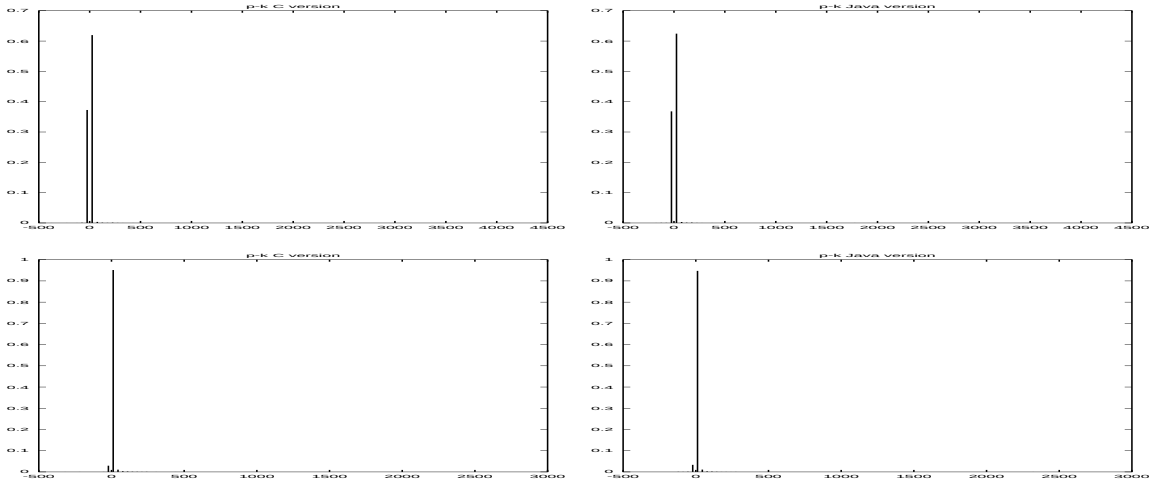

Figure 8: Histograms of error values from round trip time prediction using measurements from C (left) and Java (right) versions. All four graphs are for the p-k pair (pacers-kongo). The top pair is round trip time without the Nagle effect, the bottom is with the Nagle effect. The heavy tailed nature of the data makes it difficult to compare the predictability of the C and Java. We include only one pair of hosts for this reason, and provide CDFs for all pairs in order to evaluate the predictability of round trip time measures.

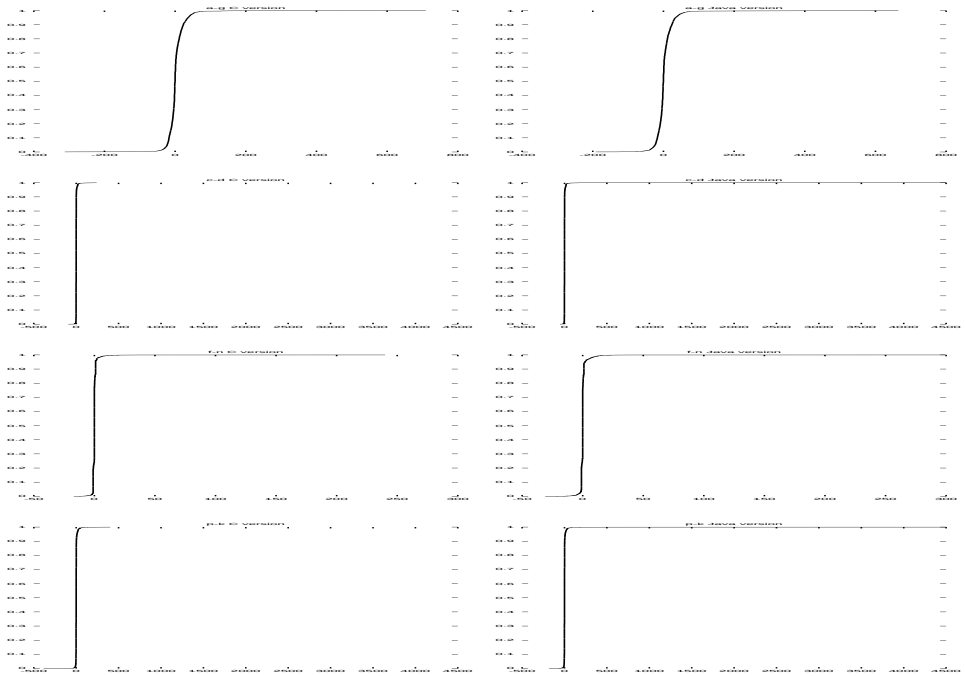

Figure 9: Cumulative distribution function (CDF) of the error values from round t rip time predictions using measurements from C (left) and Java (right) versions. The graphs indicate that the Java version of the round trip time data is as predictable as the C version.

1990. `http://www.ietf.cnri.reston.va.us/rfc/rfc1157.txt`.

[5] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributeddata-intensive applications. In *Proceedings of SC99*, November 1999.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[7] C. Granger and P. Newbold. *Forecasting Economic Time Series*. Academic Press, 1986.

[8] R. Haddad and T. Parsons. *Digital Signal Processing: Theory, Applications, and Hardware*. Computer Science Press, 1991.

[9] The cooperative association for internet data analysis – `http://www.caida.org`.

[10] The internet performance and analysis project – `http://www.merit.edu/ipma`.

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

[12] Java and High Performance Computing Group at the School of Computer Science, Telecommunications, and Information Sciences at DePaul University. `http://www.jhpc.org`.

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

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[15] J. Moreira, S. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1998. IBM Research Report 21166.

[16] The NPACI HotPage: a framework for scientific computing portals, 1999. `http://hotpage.npaci.edu/Publications/ComputingPortals99.ppt`.

[17] The network weather service home page –`http://nws.npaci.edu`.

[18] J. G. F. Panel. Report: Making java work for high-end computing. Technical Report JGF-TR-1, 1998.

[19] A. Su, F. Berman, R. Wolski, and M. Strout. Using AppLeS to schedule a distributed visualization toolon the computational grid. *International Journal of High Performance Computing Applications*, 13, 1999.

[20] 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.

[21] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 1999. available from `http://www.cs.utk.edu/~rich/publications/nws-arch.ps.gz`.