Using JavaNws to Compare C and Java TCP-Socket Performance

Chandra Krintz

Department of Computer Science and Engineering
University of California, San Diego
ckrintz@cs.ucsd.edu

Rich Wolski

Computer Science Department
University of Tennessee, Knoxville
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. However, due to the Java language implementation and functionality (portability, security, etc), it is unclear whether a Java program is able to measure and predict the network performance experienced by C-applications with the same accuracy as an equivalent C program. We provide a quantitative equivalence study of the Java and C TCP-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 via many interconnected, non-dedicated networks. Available network performance (latency and bandwidth) fluctuates over short timescales, however, 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 [2], netperf [13], pathchar [12], the Unix `ping` command, etc. as well as a raft of others currently listed with [10] and [11]). All of these tools provide an estimate of *past* performance conditions. 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 [19]. Furthermore, most of these tools use network protocols different from those used by user applications or only determine network utilization in terms of aggregate packet traffic at intermediate gateway nodes. It is difficult to translate these readings into the performance a user will experience during a remote invocation or a network download. To make an effective decision about network use, users require application level predictions of the network performance that the application or download will experience.

To solve these problems, we have previously developed the Network Weather Service [20, 19, 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[2] or administrator must install the package on any machine that is to be monitored. In an Internet computing environment, users expect to access all services through their web browsers. Rather than providing a pre-compiled binary for download, we have developed an Java applet version of the NWS monitoring and forecasting facilities. When the applet is invoked, it periodically measures the performance between the browsers into which it is loaded, and the machine running the web server that launched it. If the applet is installed at replicated server sites, the user can use the performance forecasts it generates to select the most effective server dynamically.

In this paper, we describe an implementation of NWS functionality for the Internet: *JavaNws*. We selected the Java [16, 6] language for its applet execution model as well as for its wide spread availability and use as an Internet programming language. The applet execution model, in which programs are downloaded to a user's desktop then executed locally, enables establishment of an interactive session between the desktop and the remote server machine. During such a session, the JavaNws measures network performance and displays it graphically on the user's desktop. The applet execution model also enables the user to circumvent the need to explicitly install and maintain an NWS network monitoring process. An arbitrary user can simply click on a link at a web server 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 dynamically and continuously.

However, the common Java execution environment enables portability, security, and other functionality, many times at the cost of performance. To be accepted and used in Internet computing settings, the JavaNws must report measurements of network performance that is experienced by applications that depend on the C language TCP-socket interface. For example, file downloads via HTTP and distributed applications are likely to use services built using C [9, 4, 5]. With this work, we empirically compare C and Java socket performance and show that despite any overhead imposed by the Java language design, C and Java TCP-socket implementations perform equivalently. We use the JavaNws infrastructure to collect the data for this evaluation.

In the next section we describe the implementation of the JavaNws. Section 3 details the experimental methodology we used in this study. In Section 4, 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 JavaNws Implementation

The JavaNws is a Java implementation of the NWS network measurement and forecasting subsystems (see [20] and [19] 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 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

[2] All NWS services can run with standard user privileges so any user can install the software.

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 [3, 18].

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. A complete description of the JavaNws design can be found in [14]. We next provide an overview of the JavaNws functionality.

2.1 The JavaNws Applet

The JavaNws applet establishes an interactive session with the Echo Server during which a series of communication *probes* are conducted. With each probe, measurements are taken of round trip time and bandwidth (additional detail is provided in Section 2.1.1). These experimental results are then used by a forecasting module within the applet to make predictions of the network performance that will occur at the next time step. This module is further described in the following section. The prediction and the measurement data are then visually displayed for the user as continuously updated graphs. Figure 1 provides a view of these graphs.

2.1.1 JavaNws Network Performance Measurements

To measure round trip time, a two-byte packet is exchanged between the JavaNws applet and the Echo Server during a session. To determine the extent to which the Nagle algorithm [3] effects the transfer time, two tests are performed; one with the Nagle effect off and one with it on. 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. Round trip time with the Nagle effect is reported to the user (via the display). For the results in this paper, we report data with and without the Nagle effect to show differences between the Java and C versions.

To measure bandwidth, the applet times a "long" transfer (64KB by default, although JavaNws allows the user to set the probe duration) and calculates the resulting bandwidth. This timed exchange consists of a large packet from the applet to the Echo Server and a 2 byte packet from the Echo Server to the applet acknowledging the completion of the probe. In our results section, both the C and Java versions measure bandwidth with a 64KB transfer.

2.1.2 JavaNws Forecasting Module

To make predictions of near-future performance we implemented the NWS forecasters in Java. A detailed description of the these forecasters can be found in [19]. In short, this module consists of a set of independent forecasting algorithms [7, 1, 8], each of which produces a one-step-ahead forecast from a given time series. At each time step, the measurement data taken by the applet is 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 mean square forecasting error (MSE) associated with each forecasting

[3] 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.

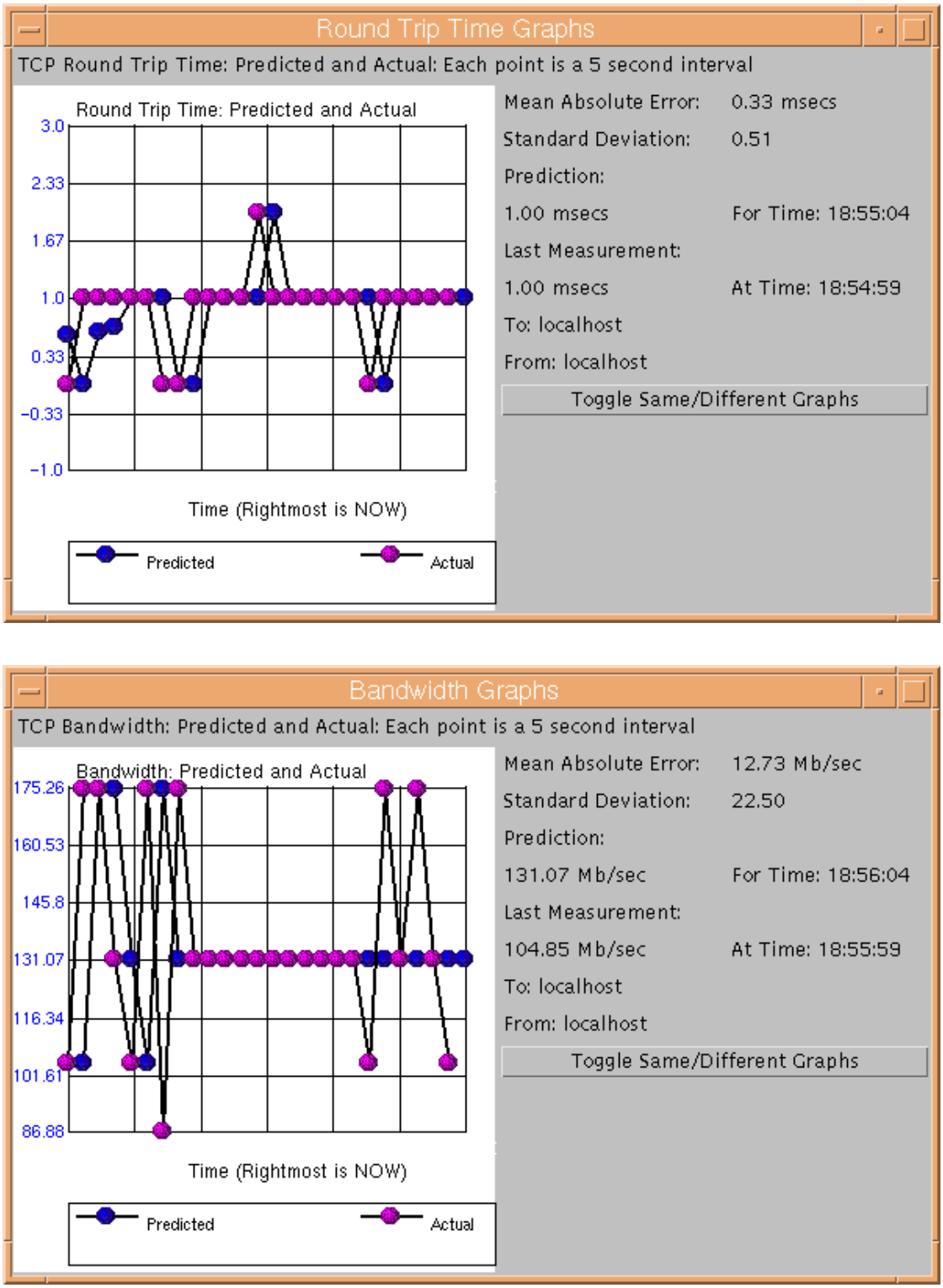

Figure 1: A view of the JavaNws round trip time (TOP) and bandwidth (BOTTOM) graphs in action. Dark (blue) points are predicted values and light (pink) points are the actual measurements. The right-most point in each graph is the value predicted to occur in the next (future) time step. Where there appears to be only a single point at a given time step, the predicted value is the same as the actual measurement (the measurement was correctly predicted).

algorithm is accumulated for every measurement. 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 JavaNws prediction. Note that this forecasting module keeps a separate time series for round-trip time, Nagle-impeded round-trip-time, and bandwidth.

2.2 Validating The JavaNws

In the remainder of this paper, we validate the JavaNws by empirically comparing C and Java TCP-socket interfaces. That is, we show that the JavaNws measurement and prediction of network performance between the desktop and a remote server is as accurate as its C counterpart and that any overhead imposed by Java programming language does not effect the Java TCP-socket performance. Our results in the following sections indicate that JavaNws is able to measure and predict the network performance available for download and distributed execution by applications written using the C TCP-socket interface. We use the JavaNws infrastructure to collect the necessary data for this study as described in the following section.

3 Experimental Methodology

To empirically compare the C and Java TCP-socket interfaces, we extended the JavaNws to use the Java Native Interface (JNI) to invoke a native C language method each time it takes a measurement. Once a transfer has been completed by the Java version (the Java version measurements have been taken and the socket used in the exchange has been torn down), the applet calls a C native method that repeats the communication protocol to collect the C version measurements [4]. All experiments were performed at approximately 12 second intervals over a 24 hour period on weekdays between various machine pairs.

We chose pairs of machines based on the type of network interconnecting them to observe Java and C performance under varying conditions. Data for four such links, representative of the overall set, is included in this paper. Table 1 gives the location of machines for each pair of hosts, the predominant network technology separating the two, and the version of Java interpreter that was used to execute JavaNws 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. The "standard" Internet connectivity we observed in this study is non-dedicated common-carrier. We also show the types of machines and operating systems we used in Table 2 as end-to-end performance can be influenced by the type of software and hardware at the end-points.

Graphs of the 24-hour, raw data measurements taken by the C version and the Java version for each pair of hosts are shown in Figures 2 (bandwidth), 3 (round trip time) and 4 (round trip time with the Nagle effect). The left graph in each row is the data collected by the C version; the right collected by the Java version.

[4] We also performed the measurements in the opposite order (the C version first and then the Java version) and found similar results.

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

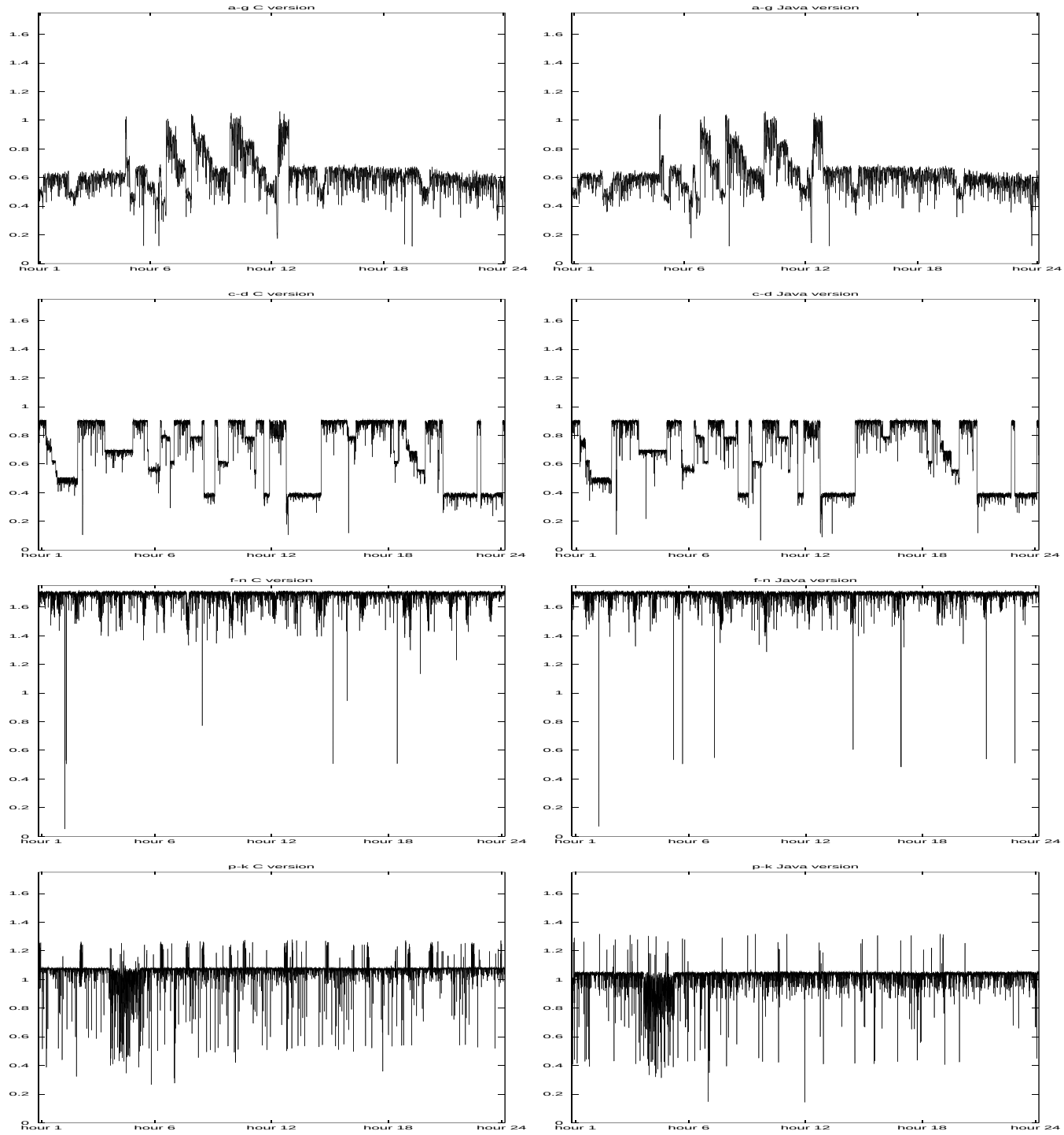

Figure 2: Raw 24-hour bandwidth data. Left graph for each pair (row) is the C version, the right graph is the Java version. Each host pairs is shown. The x-axis is time and the y-axis is bandwidth in Mb/s.

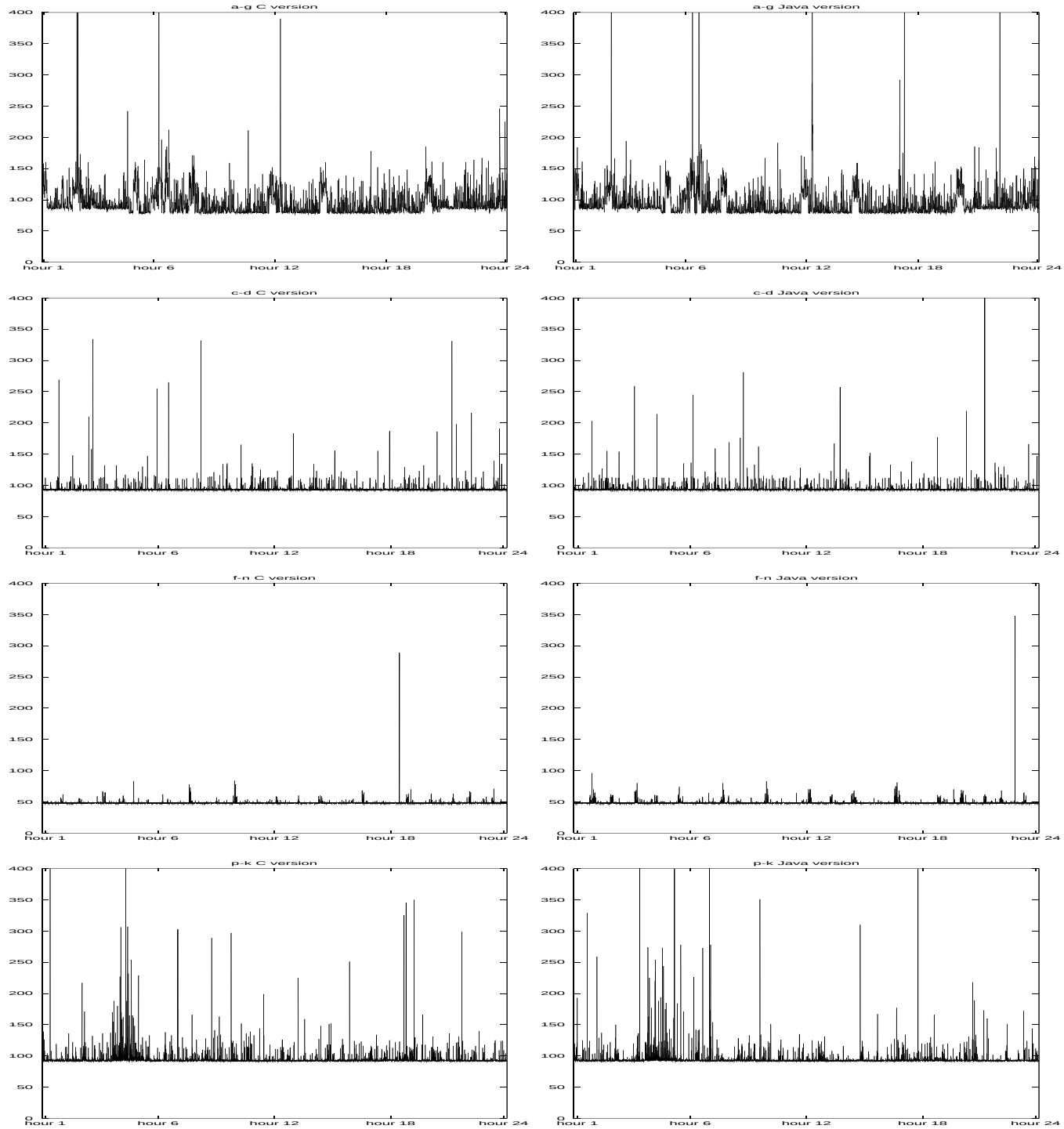

Figure 3: Raw 24-hour round trip time data. Left graph for each pair (row) is the C version, the right graph is the Java version. Each host pairs is shown. The x-axis is time and the y-axis is round trip time in milliseconds.

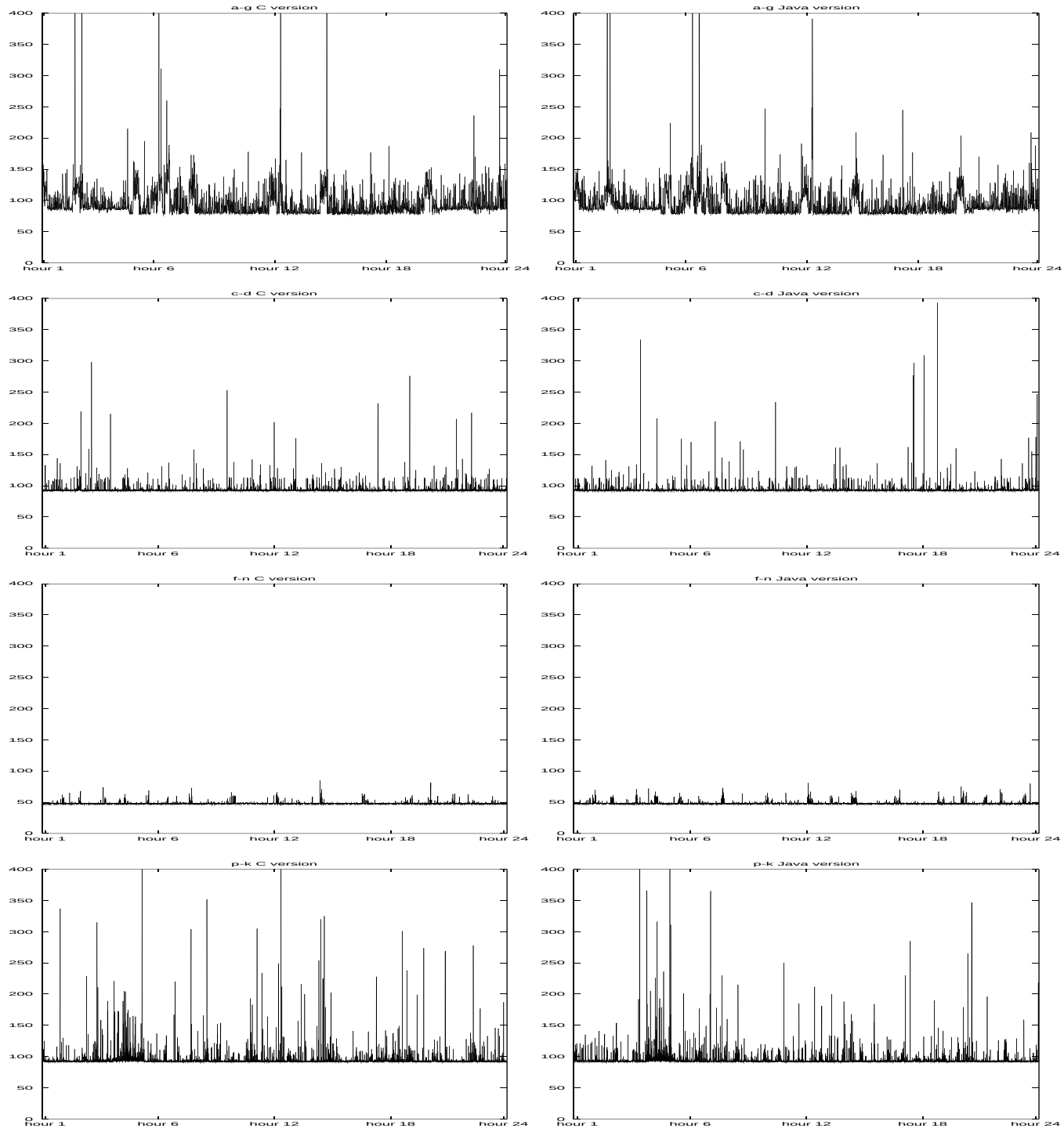

Figure 4: Raw 24-hour round trip time (with the Nagle effect) data. Left graph for each pair (row) is the C version, the right graph is the Java version. Each host pairs is shown. The x-axis is time and the y-axis is round trip time in milliseconds. The Nagle effect is a TCP optimization that delays small packets in attempt to combine many into a single large transfer. If no other packets are sent, the packets are forced to wait unnecessarily.

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

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, the underlying network technology, the software installed at the end points, and the machines involved in the transfer. To prevent network congestion, however, the underlying communication protocol (TCP/IP in this case) adapts transfer rates in response to dynamically changing traffic patterns. As such, a controlled quiescent network may cause C and Java to appear to have the same performance, but that performance may differ when they are used to traverse networks experiencing load. Since the load is varying quickly [15] it is 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 very similar. The round trip time variance for Java is greater than that for C in the numbers presented. This is due to a small number (< 6 measurements in 24 hours) of very large measurements that result from random catastrophic network events that occur for which the experiment times out. It is our experience that this also occurs for the C measurements but less frequently in the data provided in this paper. Such values significantly effect the variance but since they occur so infrequently the

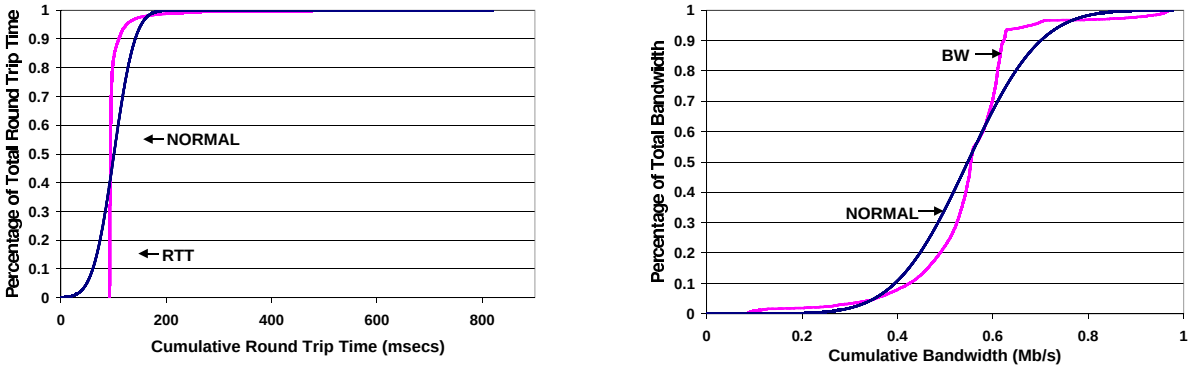

Figure 5: Comparison of the cumulative distribution functions (CDFs) for round trip time (left) and bandwidth (right) measurements for the c-d data set resulting from the Java version. The CDF of each resource is compared to the normal distributions derived from the mean and variance of the sample. The graphs for the C version are indistinguishable.

mean is unaffected. Unfortunately, since network performance data is not well approximated by Normal or other forms of exponential distributions, it is difficult to determine the statistical significance of this comparison rigorously. If, for example, the distributions are heavy-tailed, it is possible that extremely large samples must be observed before a comparison of the moments can be exposed to a rigorous statistical test. As an example of how poorly a Normal approximates the distributions we observe, Figure 5 compares CDFs for the observed c-d data set (latency and bandwidth) with Normal curves having the same sample mean and variance.

4.2 Comparison based on Regression Coefficients

Moment-based comparisons assume that the time at which each sample is gathered is irrelevant since the sample statistics are not weighted with respect to time. That is, it is possible for the means and variances to appear to be the same, but for a pairwise comparison to reveal a difference between samples gathered at approximately the same time. To capture this notion of time-dependent comparison, we can calculate the regression coefficient between samples. We pair together values from each sample 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 closely the derived multiplicative constant is to 1.0 for the different traces. Table 5 shows the results. A value near 1.0 indicates that, on the average, multiplying the C value by 1.0 yields an accurate prediction of the corresponding Java value.

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 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.

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

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. That is, at various points in time, either the C or the Java measurement (but not both together) was affected by an interruption in network connectivity. In order to rectify this problem and report a value unaffected by outliers, we removed obvious outliers (pairs of measurements where one was 2 or 3 orders of magnitude bigger than the other) 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 we examined in our study. 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 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 is important to ensure equivalent performance from the C and Java versions. In the JavaNws implementation we have developed, we use the

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 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 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 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 pair of long integers (one for the number of seconds since 1970 and one containing the number of microseconds since the last measured second). 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 can frame the problem of determining equivalence in terms of the degree to which one set of measurements can be used to predict *future* values of the others. If the technologies are equivalent in their predictability, the observed difference in variance can be treated as random measurement error.

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. That is, regardless of statistical significance, the prediction error is what we observed under actual load conditions. The consistency with which we observed it is the basis for our conclusions.

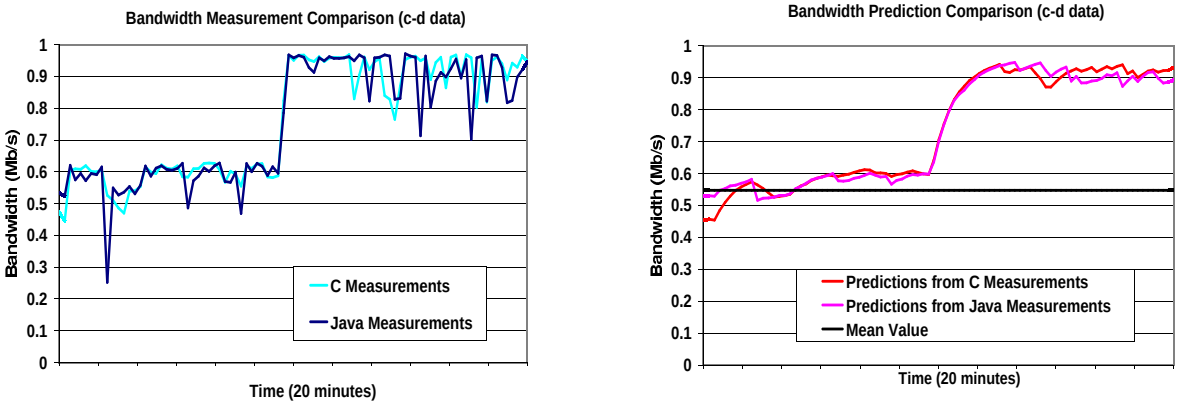

Figure 6: Comparison of C and Java *actual* (left) and *predicted* (right) bandwidth measurements over a 20 minute interval. The x-axis is time (20-minutes) and the y-axis is Mb/s. The data is taken from the cross-country, c-d, 24-hour data used throughout this study. The right graph of the figure exemplifies the smoothing effect prediction has on the data. Observation of the predicted values indicates that the C and Java data sets are quite similar. This is less clearly observed from the actual measurements in the left graph.

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 [19]). 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. In addition, using prediction based on actual values also ”smoothes” the data by eliminating outlying values that must be treated as random or catastrophic network events. Figure 6 shows a 20-minute snapshot of data in this time series format of C and Java bandwidth measurements (left graph) and the NWS predicted values (right graph) computed using C and Java measurements. The data is taken from the 24-hour c-d data. The right graph of the figure exemplifies the smoothing effect prediction has on the data. Observation of the predicted values indicates that the C and Java data sets are quite similar. This is less clearly observed from the actual measurements in the left graph.

5.1 Histograms and CDFs of Error Values

In addition to the observation of time series from forecasted values, the predictability of two samples can be compared using a histogram of the error values. The error value is the difference between the NWS predicted value and the actual value that it predicts for each time step. For both the C and Java versions of each set of predicted measurements, we form a histogram of 100 bins spanning the difference between the minimum and maximum measured values. Figures 7, 8, and 9 contain the histograms for each pair of hosts for bandwidth (7), round trip time (8), and round trip time with the Nagle effect (9). The left graph is the C

version and the right is the Java version. Clearly, the error histograms are almost identical. [5]

Histograms are useful for visualizing distributional data but prone to variation due to bin size. Indeed, the predictability for the round trip time (Figure 8 and 9) measurements are difficult to compare using this format. This difficulty is due to the the presence of infrequently occurring values that we cannot rule out as outliers. 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 generated graphs of the data in cumulative distribution functional (CDF) form. Figures 10, 11, and 12 contain the CDFs of the prediction errors for bandwidth and round trip time, respectively, for each host pair and version. CDFs indicate the percentage of error values that are below a given value. They enable us to compare two data sets for equivalence in terms of this percentage. For example, in the first row of Figure 10, both the C and the Java version CDFs indicate that 55% of the error values fall at or below zero. When the C and Java versions of the CDF graphs were overlayed, one masks the other almost completely indicating the same findings as those from the histogrammed values: the two data sets are equally predictable.

5.2 Empirically-derived Confidence Intervals

Another technique we can use to compare two data sets is to determine if the confidence interval for each prediction matches between the two series in question. It may be that the pairs of values differ, but that they must fall within the same range when predicted. If they do, then we conclude that the series are equivalent based on their predictability.

Since it is likely that the series of forecasting errors generated for each measurement trace is non-stationary, we are unable to supply a rigorous definition of a confidence interval. Instead, we calculate a conditional confidence interval empirically, at each point in a measurement trace, based on the forecasting errors associated with the values proceeding that point. That is, we define $CI_t(M)$ to be the forecasting confidence interval at time t within a measurement trace M. $CI_t(M) = (f_t - 2\times\sqrt{MSE_t}, f_t + 2\times\sqrt{MSE_t})$ where f_t is the forecast of the measurement value occurring at time t, and MSE_t is the sample mean of the squared forecasting errors occurring before time t in trace M.

In this formulation, each f_t is treated as a conditional expectation of the measurement it forecasts. The $\sqrt{MSE_t}$ is analogous to the conditional sample standard deviation of the forecasting error series. As such, it resembles the formula for calculating a 95% confidence interval for a Normally distributed sample which is $(m - 1.96 \times sd, m + 1.96 \times sd)$ for a population sample with mean m and standard deviation sd. We make no claims about the optimality or bias of our formulation. Rather, we note that it has proved a useful engineering approximation for measurement series gathered by the NWS.

By establishing confidence boundaries using two standard deviations from the predicted values, we create two new time series between which we postulate that 95% of the measurements will fall. In addition, if at least 95% of the Java measurements are captured between the C empirically-derived confidence boundaries, and vice versa, then Java and C are equally predictable. That is, if this is the case then it does not matter which language was used to generate the predicted values or measurements since they are indistinguishable within engineering tolerances (95%).

Table 7 shows the empirically derived confidence intervals for the data sets and the percent of measurement values captured by each language. Indeed, the data sets are indistinguishable and one could be used to predict the other. That is, the data sets are equivalent within common engineering thresholds and C and Java TCP-socket implementations are equivalent. This point bears repeating. If 95% of the Java measurements

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

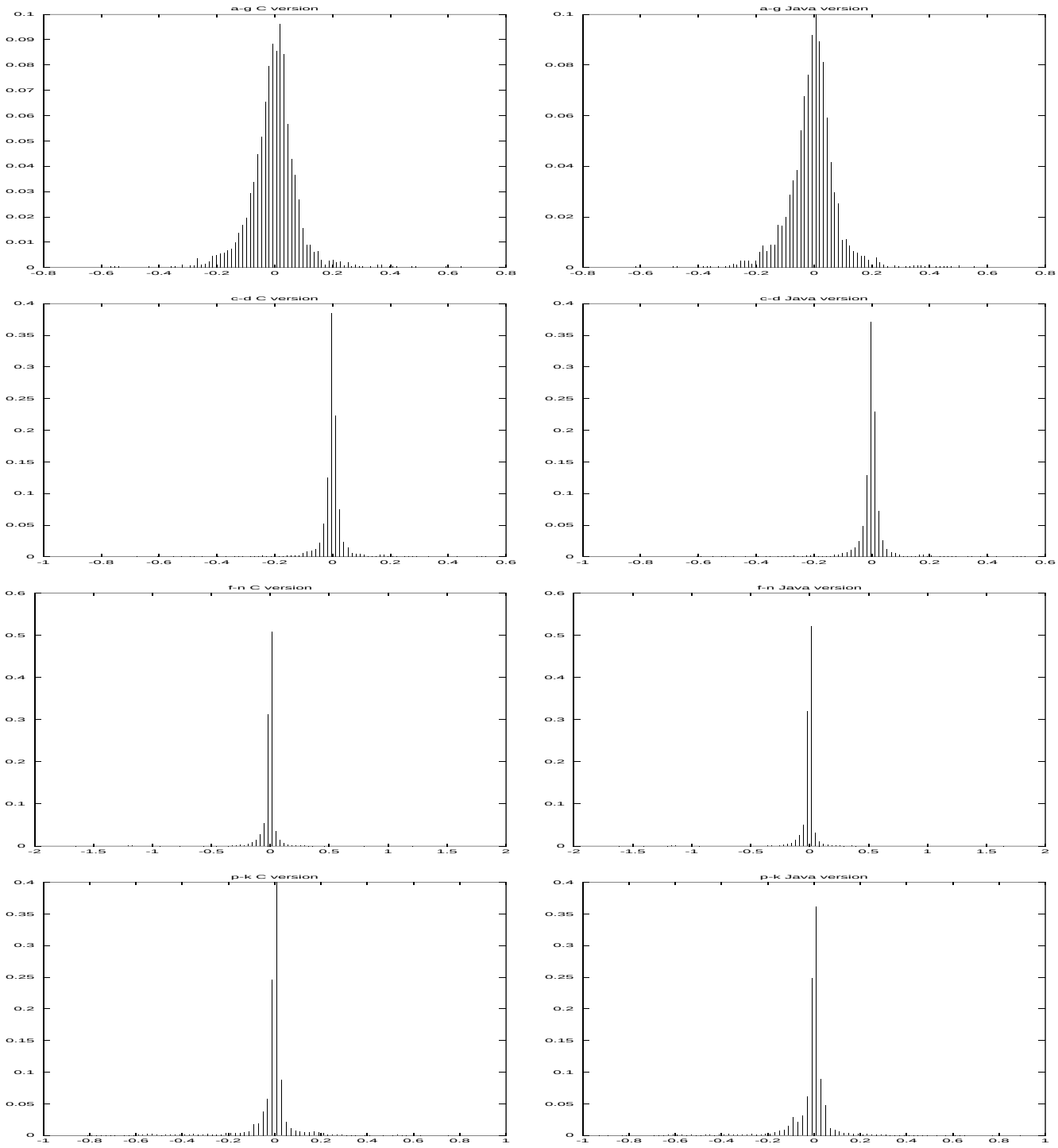

Figure 7: Histograms of error values from bandwidth prediction using measurements from C (left) and Java (right) versions. The predictability of two samples can be compared using a histogram of the error values. The error value is the difference between the NWS predicted value and the actual value that it predicts for each time step. For both the C and Java versions of each set of predicted measurements, we form a histogram of 100 bins spanning the difference between the minimum and maximum measured values. The graphs indicate that the Java version of the bandwidth data is as predictable as the C version.

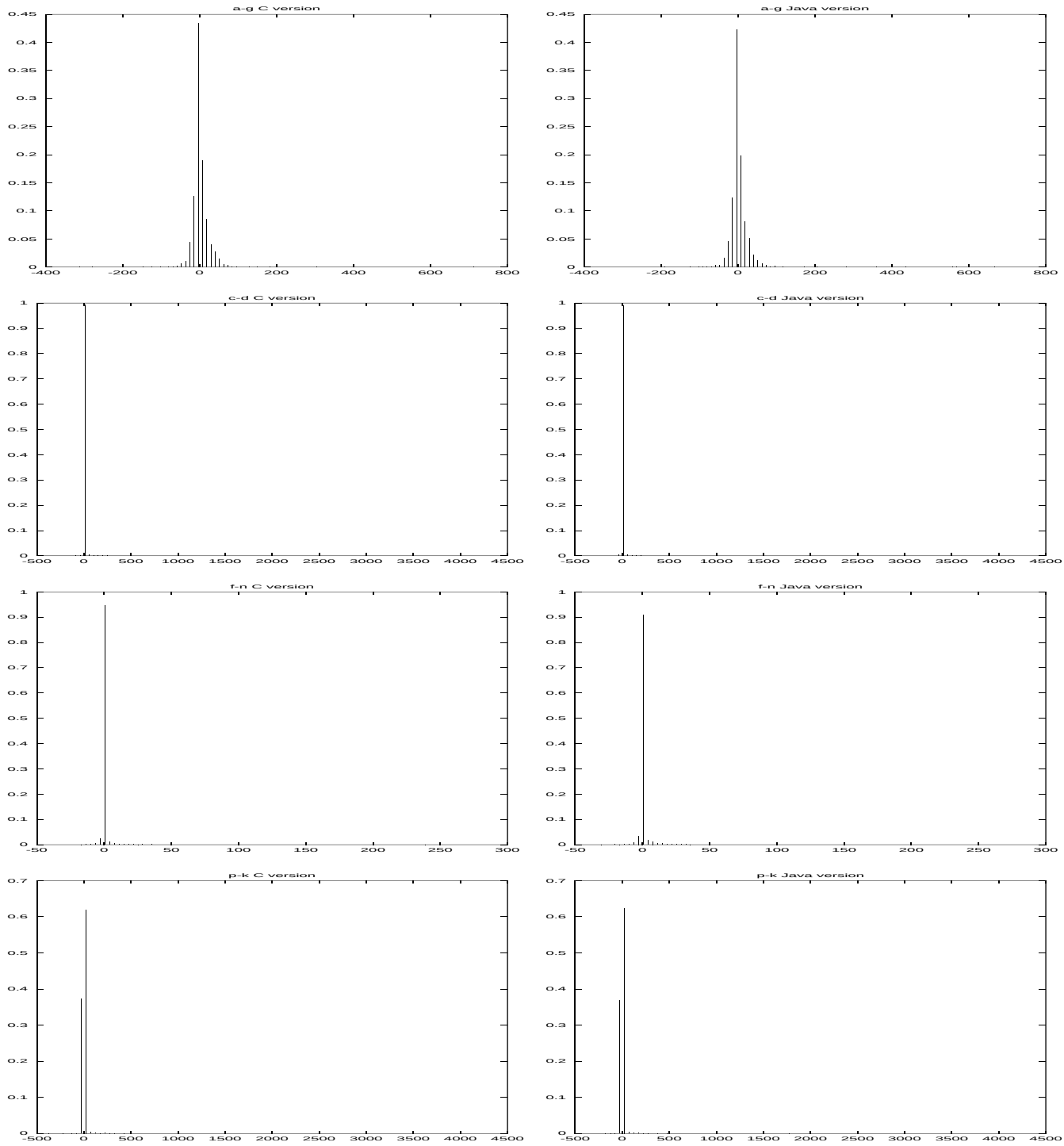

Figure 8: Histograms of error values from round trip time prediction using measurements from C (left) and Java (right) versions. The predictability of two samples can be compared using a histogram of the error values. The error value is the difference between the NWS predicted value and the actual value that it predicts for each time step. For both the C and Java versions of each set of predicted measurements, we form a histogram of 100 bins spanning the difference between the minimum and maximum measured values. The graphs indicate that the Java version of the round trip time data is as predictable as the C version.

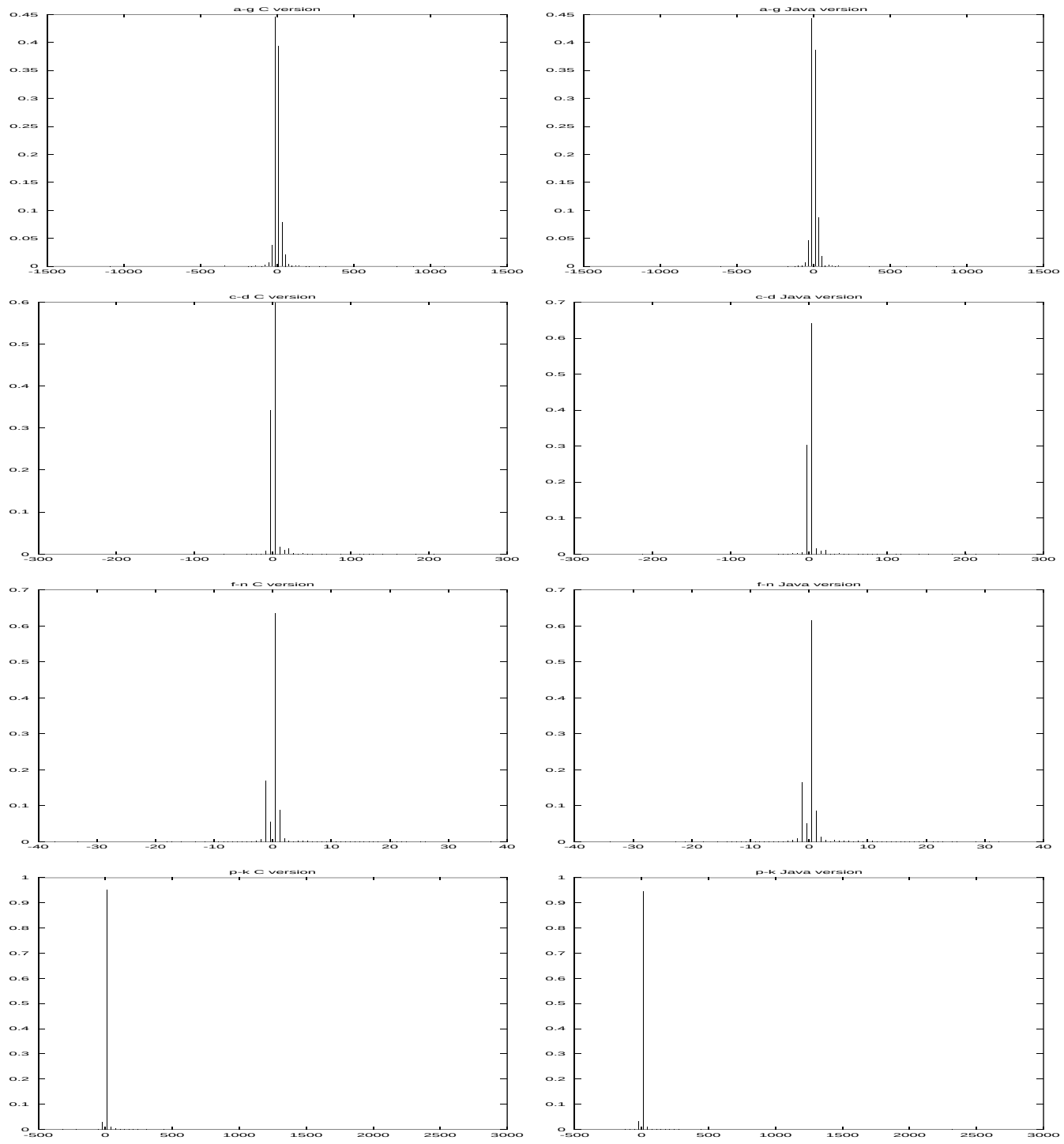

Figure 9: Histograms of error values from round trip time (with the Nagle effect) prediction using measurements from C (left) and Java (right) versions. The predictability of two samples can be compared using a histogram of the error values. The error value is the difference between the NWS predicted value and the actual value that it predicts for each time step. For both the C and Java versions of each set of predicted measurements, we form a histogram of 100 bins spanning the difference between the minimum and maximum measured values. The graphs indicate that the Java version of the round trip time data (with the Nagle effect) is as predictable as the C version.

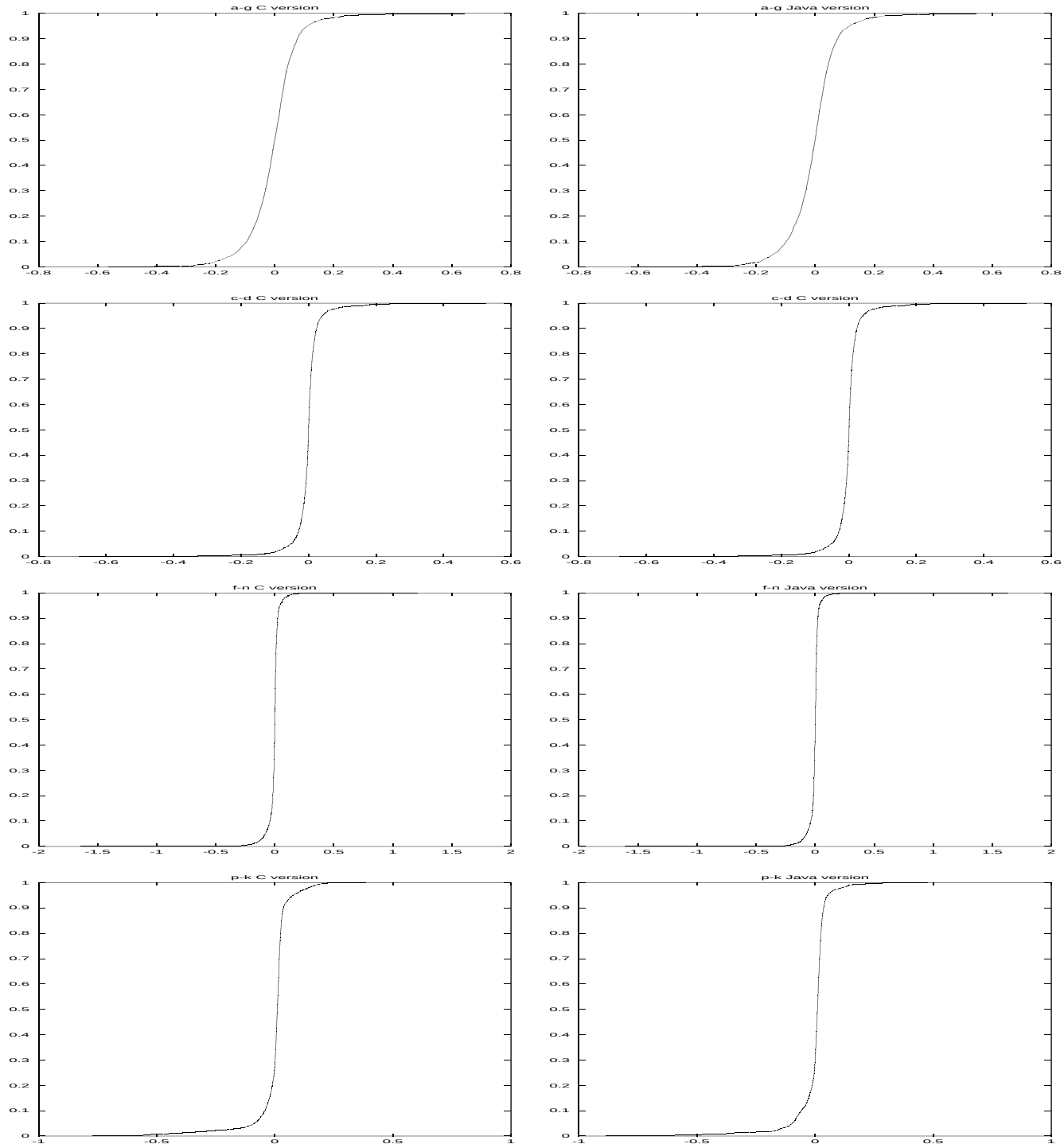

Figure 10: Cumulative distribution function (CDF) of the error values from bandwidth predictions using measurements from C (left) and Java (right) versions. An alternative method for comparing predictability of two data sets is to visualize the cumulative distribution of each set. CDFs indicate the percentage of error values that are below a given value. They enable us to compare two data sets for equivalence in terms of this percentage. The graphs indicate that the Java version of the bandwidth data is as predictable as the C version.

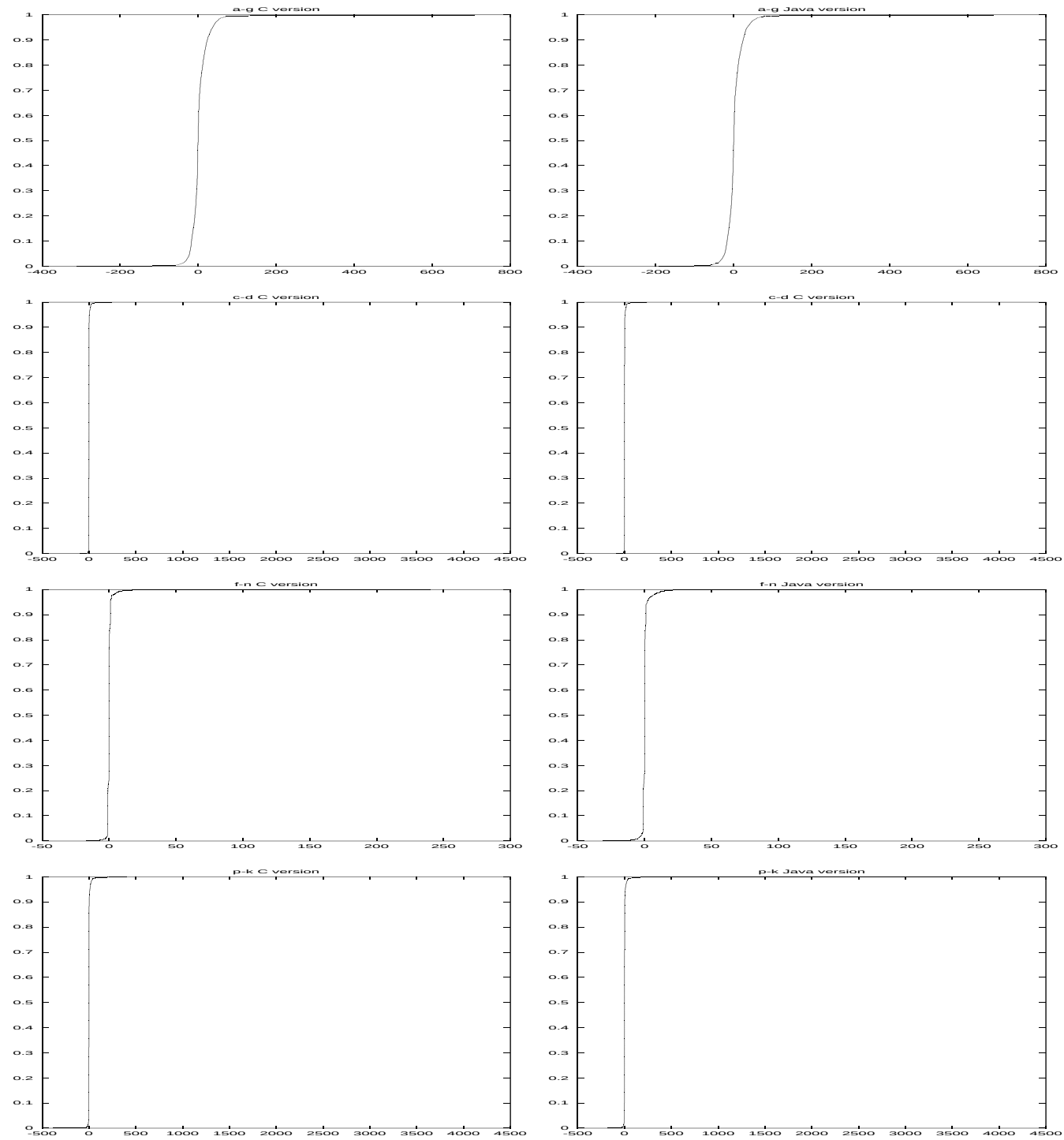

Figure 11: Cumulative distribution function (CDF) of the error values from round trip time predictions using measurements from C (left) and Java (right) versions. An alternative method for comparing predictability of two data sets is to visualize the cumulative distribution of each set. CDFs indicate the percentage of error values that are below a given value. They enable us to compare two data sets for equivalence in terms of this percentage. The graphs indicate that the Java version of the round trip time data is as predictable as the C version.

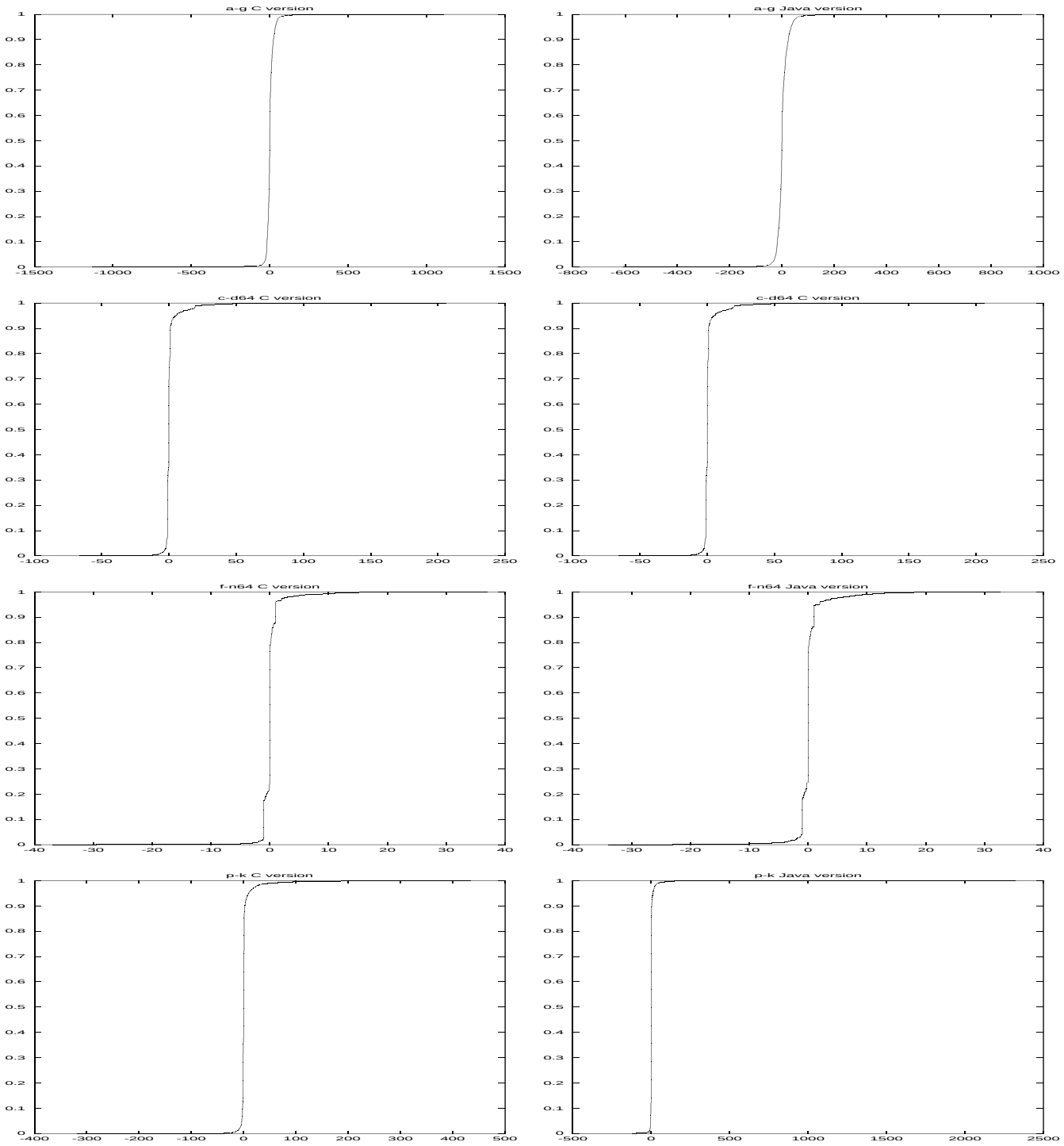

Figure 12: Cumulative distribution function (CDF) of the error values from round trip time (with the Nagle effect) predictions using measurements from C (left) and Java (right) versions. An alternative method for comparing predictability of two data sets is to visualize the cumulative distribution of each set. CDFs indicate the percentage of error values that are below a given value. They enable us to compare two data sets for equivalence in terms of this percentage. The graphs indicate that the Java version of the round trip time (with the Nagle effect) data is as predictable as the C version.

Table 7: 95% Empirically-derived (E-d) confidence intervals and the percent of capture for each language. The values in each (row,column) indicate the percentage of measurement values (generated using the language indicated by the column header) that fall within (are "captured") by the 95% empirically-derived confidence interval for the language indicated by the row header. For example, using the fifth row and second column of data: 97% of Java measurements (column header) are captured by the 95% e-d confidence interval for the C language (row header)

		Round Trip Time		Bandwidth	
Host Pair	Language	Pct. of C meas. Captured	Pct. of Java meas. Captured	Pct. of C meas. Captured	Pct. of Java meas. Captured
a-g	C	98	98	95	95
	Java	98	98	95	95
c-d	C	97	98	97	96
	Java	98	98	97	97
f-n	C	98	97	97	98
	Java	97	98	98	97
p-k	C	98	98	96	95
	Java	99	99	97	94

fall within the corresponding CI values generated for a C measurement trace, and vice versa, the series are interchangeable for the purposes of making performance predictions in an engineering setting.

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.

As part of future work, we plan to investigate the effect of using Just-In-Time compiled and JNI implemented Java TCP/IP socket measurement routines. In addition, since the JavaNws is used on a single server by multiple clients, we will also determine the amount of latency that is imposed by the server load necessary to respond to client requests. By incorporating JavaNws access to an NWS server, we will be able to monitor and predict server load. Such measurements can be displayed in a JavaNws graph to enable the user to visualize both the network and server CPU performance.

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. To measure and predict network performance, the 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.

To validate that JavaNws can be used to measure and predict network performance available to any download or application written using the Java *or* C TCP-socket interface, we provide a quantitative study of the performance differences between Java and C socket implementations. 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 these sample-based techniques indicate that the data collected by the Java version is equivalent (within engineering tolerances) to that by the C version.

We then compare the predictability of the JavaNws data to that of the equivalent C version. We compute error values (differences between predicted and actual values) using the NWS forecasting algorithms and analyze both the error and predicted values as time series. We then construct histograms and cumulative distribution functions (CDFs) that indicate very little differences between the C and Java error value series. Finally, we describe a novel and empirical extension to the statistical confidence interval technique. With empirically-derived confidence intervals we are able to show that predictions made using Java measurements are indistinguishable from those made using C measurements. Our detailed analysis establishes that the C and Java TCP-socket interface implementations are equivalent.

References

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

[2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp), May 1990. `http://www.ietf.cnri.reston.va.us/rfc/rfc1157.txt`.

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

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

[5] 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.

[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] Hypertext transfer protocol – HTTP/1.1, Jan. 1997. `http://www.ics.uci.edu/pub/ietf/http/rfc2068.txt`.

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

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

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

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

[14] C. Krintz and R. Wolski. Javanws: The network weather service for the desktop. In *JavaGrande*, June 2000.

[15] W. e. a. Leland. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, February 1994.

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

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

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

[19] 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.

[20] 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.