

# Network Aware Data Transmission with Compression

Ningning Hu \*

*Computer Science Department  
Carnegie Mellon University  
(hnn@cs.cmu.edu)*

**Abstract** *Network aware application* can achieve better performance by dynamically adapting to network service changes. The key question for network aware application development is how to obtain information about the performance of different system module. In this paper, we consider an important category of network aware application – *Compressed Data Transmission*. Compression can reduce network transmission time by reducing the size of data to be transmitted, but on the other hand it increases local processing overhead. The tradeoff between increased network processing and decreased local processing is critical to application’s decision on how to transfer data. In this paper, we present our model to make such decision and discuss the methods of detecting network resources and predicting compression performance parameters. Experimental data on local testbed is presented to evaluate our methodology. We also discuss an improved model on how to deal with the overlap between network transmission and local processing, which has the potential to improve the application performance.

## 1 Introduction

The Internet is well known for its unpredictable performance, people expect to experience different level of Internet service in different places and at different times. When users can choose from a number of sites providing the same service, which is often the case on current Internet, they generally have no idea which one is the best to choose. *Network Awareness* is one of the techniques to deal with these problems. It enables an application to change its behavior according to network performance, allowing it to achieve consistent performance over a diverse sets of networks and under a wide range of network conditions.

In this paper, we focus on an important category of network aware application – *Compressed Data Transmission*. This type of application has the ability to

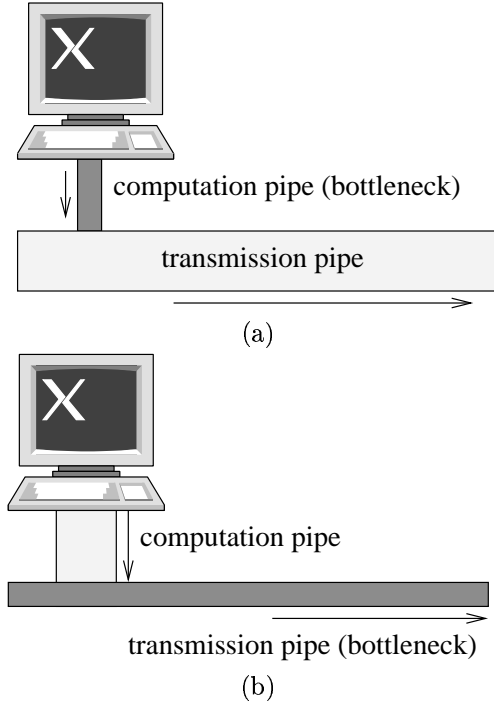
compress data before sending it out with the potential to reduce network transmission time and reduce the total application elapsed time. An example scenario in which *Compressed Data Transmission* can be useful is as follows. Suppose we are leaving for a conference by air. Just before boarding, we want to transfer a big file back to our office machine through a wireless LAN. But at the same time, another plane arrives, a lot of people get off. They start using all kinds of mobile communication devices to send and receive voice and digital messages, which saturates our wireless transmission channel. Under this condition, compression may be an important tool to help us in finishing our work without missing the flight.

Compression can reduce the transmission time by reducing the amount of data to be transferred. But it also increases the local processing time by introducing the compression overhead. Whether or not an application can benefit from data compression depends on the tradeoff between the reduction of network transmission time and the increase of local processing. In another way, we can think the whole procedure of data transmission as composed by two data flow pipes (Fig. 1). The first one is the data flow from local host to network interface. The rate of this flow is determined by the host processing capability. The other pipe is the data flow on the network. The rate of this pipe, that is, the data transmission rate on the network, is determined by the available bandwidth. The performance of the two pipes together determines the performance of the application. For example, in Fig. 1(a), the available bandwidth is large enough, and the network pipe could transmit data faster than the host pipe. Under this condition, the performance of the application is largely determined by the host pipe rate. Similarly, in Fig. 1(b), the application performance is determined by network pipe rate. Determining which one is the bottleneck during the execution, need some techniques to calculate the concrete network transmission time and local computation time.

In this paper, we describe our methods to detect and predict network performance and compression

---

\*Ningning Hu is advised by Prof. Peter Steenkiste.



Data flow pipes involved in *Compressed Data Transmission*. In (a) the bottleneck is on the local computation pipe, while in (b) the bottleneck is on the network.

Figure 1: Data flow pipes

overhead. A simple model using these techniques to prediction the application performance is presented. Experiment is carried out to evaluate the performance of the simple model. We will see that, although in most cases the simple model could make the right judgment for application, the difference between predicted values and measured values is significant sometimes. Analysis of the prediction error tells us that the overlap between different types of processing modules must be considered to make a correct judgment. Based on these considerations, we propose an improved model, which explicitly considers the effect of different performance bottleneck.

This paper is organized as follows. Section 2 abstracts *Compressed Data Transmission* into a simple application prototype, and discusses the detailed techniques to predict compression performance and network performance. In Section 3, experimental data and analysis are presented to show the application performance. Section 4 is an extended discussion of the experimental data, and the modeling of the overlap between network data transmission and local processing, which can improve the prediction performance. Section 5, 6 and 7 talks about the related work, conclusion and future works, respectively.

## 2 Architecture

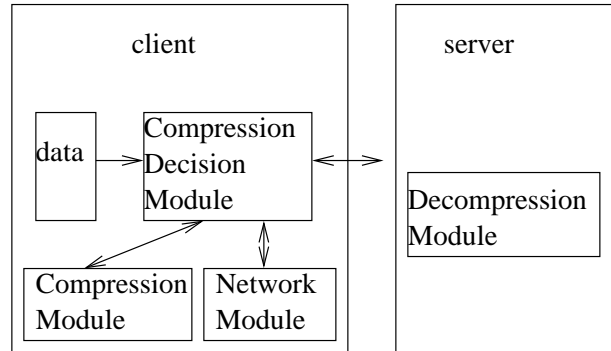


Figure 2: Application architecture

The application used in our study is a simple data transmission application (Fig. 2). It has two functions: compression and transmission, with the cooperation of the following three modules:

1. *Compression Module*: provides the functionalities for data compression, and is also responsible for providing compression algorithm related parameters, i.e., *compression speed* and *compression ratio*.
2. *Network Module*: monitors the network performance properties and provides necessary information for the application, such as *available bandwidth*.
3. *Compression Decision Module*: makes the final decision whether the application can benefit from data compression.

To send out a large amount of data, the application first splits it into multiple chunks, and processes each chunk separately. For each chunk of data, it first calculates the total execution time with and without compression using the parameters from *Network Module* and *Compression Module*, compares them and makes the decision how to transfer the data. *Network Module* provides the *available bandwidth* information, which is used to compute the network transmission time; *Compression Module* maintains the empirical data related with the specific compression algorithm – *compression speed* and *compression ratio*, which are used to calculate the local compression time.

If the application decides not to use compression, the original data will be simply sent out. Otherwise, it will be "forwarded" to the *Compression Module*, where it is compressed, before being sent to the server. During that procedure, a simple application

```

repeat {
    COMPRESSED_DATA_TRANSMISSION(data);
} until (all data is sent out);

COMPRESSED_DATA_TRANSMISSION(data)
{
    compr_speed = GET_SPEED();
    compr_ratio = GET_RATIO();
    cur_bw = GET_CUR_BW();
    if (COMPRESSED_TRANSFER(data, cur_bw,
        compr_speed, compr_ratio)){
        COMPRESS(data, &compr_data);
        SEND(compr_data);
    }
    else {
        SEND(data);
    }
}

```

Figure 3: Application pseudocode

protocol is used to tell the server the transmitted data needs decompression. The pseudocode for the application is shown in Fig. 3.

## 2.1 Compression Decision Module

Different applications may use different ways to make decision about whether or not to use compression. Some applications may choose to use compression as long as the total execution time for the data transfer with compression is smaller than transmission time without compression, with the objective of reducing the load on the network. Other applications focus on achieving best data quality, so they try to avoid using compression as long as the data transmission without compression is below some threshold, since some compression algorithm may lose information,

No matter what policy the application uses, it must calculate the total execution time for both the compressed mode and the uncompressed mode. In this paper, for the uncompressed mode, the total execution time is simply the data transmission time, which can be computed as:

$$comm\_time = \frac{data\_size}{available\_bandwidth} \quad (1)$$

where *data\_size* is the size of the data to be transmitted by the application, and *available\_bandwidth* is the available bandwidth of the network link.

The overhead involved in the data transmission with compression is computed as:

$$total\_time = compr\_time + send\_time \quad (2)$$

$$compr\_time = \frac{data\_size}{estimated\_compr\_speed} \quad (3)$$

$$send\_time = \frac{compr\_size}{available\_bandwidth} \quad (4)$$

$$compr\_size = \frac{data\_size}{estimated\_compr\_ratio} \quad (5)$$

where *estimated\_compr\_speed* and *estimated\_compr\_ratio* are the two parameters provided by *Compression Module*.

The above methods to predict the application performance shown in formula (1) - (5) is denoted as *simple model* in the following sections.

## 2.2 Compression Module

Our prototype uses the general purpose lossless compression algorithm - *gzip*[7] in the *Compression Module*. *Gzip* is a standard compression utility commonly used on Unix operating system platforms. It does not consider domain specific information and uses a simple, bitwise algorithm to compress file. It can work in stream mode, that is, at the time that the compression starts, not all data has to be available. In our implementation, we simply incorporate the *gzip* library into our application code. The performance parameters that this module needs to provide are *compression speed* and *compression ratio*. We use empirical method to get their value; see Section 3.2 for details.

## 2.3 Network Module

We use Remos in the *Network Module* to monitor and predict network performance. In the following, "network performance" means *network bandwidth available for the application*.

Remos (Resource Monitoring System)[3, 4, 15] is a network performance middleware service developed at CMU. It provides a scalable, flexible and portable network monitoring system for applications in distributed computing environments. Remos is composed of two parts: *Modeler* and *Collector*. The *Modeler* implements the Remos API, which enables applications to communicate with the *Collector*, query the interested information, and transform the data from the *Collector*. The *Modeler* also integrates some prediction services [4], allowing history-based data collected across the network to be used to generate the

predictions needed by a particular user. The *Collector* is responsible for the network performance information collection, using SNMP[18, 19] or benchmarks. The network performance information includes network topology, link latency, link capacity and link available bandwidth. Several types of collectors are implemented in Remos. They are the *SNMP Collector*, *WAN Collector*, *Bridge Collector* and *Master Collector* [15]. Different collectors work in different network environment and provide different monitoring methods.

Our experiments are carried out on a local LAN testbed. We use the *SNMP Collector*[15] to get the available bandwidth information. In a LAN, the *SNMP Collector* depends on SNMP agents[19] to collect information. SNMP agents can provide the information about the total amount of input data and output data from each network interface. By keeping track of these values, the *SNMP Collector* can estimate the average throughput of the link between two end hosts during the measurement period. Together with the knowledge of the link capacity, which can also be obtained from SNMP agents, it will be able to figure out the available bandwidth. Fig. 4 illustrates this method.

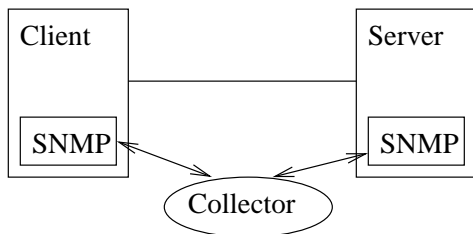


Figure 4: Available bandwidth prediction by Remos SNMP Collector

To predict the available bandwidth for the near future, the *SNMP Collector* periodically queries SNMP agents and keeps a record history (associated with a *history period* time slot). It uses the average of the history record in the history period as the prediction for available bandwidth.

### 3 Experiment on the Simple Model

In this section, we discuss our evaluation of the performance of the simple model. We first give some data about the performance of compression module and network module, then discuss the experiments for data transmission with and without compression. We will see that the simple model could give very

accurate judgment on whether or not to use compression. But we also notice the significant difference between the predicted values and the measured values, which we think is due to the defect of the simple model. In Section 4, we explain the causes of the prediction error and propose an improved model for making predictions.

#### 3.1 Experimental Setup

Experiments are carried out on our local lab testbed. Fig. 5 shows our experiment configuration. There are four host machines, the two slashed squares are the application client and the application server, and the two grid squares are the competing client and the competing server. The two circles are routers, and the link between the routers is the bottleneck link in this simple network; its nominal transmission capacity is 10Mbps. The application client and the application server are executed on two Digital Unix machines. The competing client and the competing server are used to create the competing flow so as to change the available bandwidth on the link. The application uses a single TCP connection to transfer data. While the traffic generated by the competing hosts (also called *competing flow*) uses UDP packets since it is more accurate to predict the available bandwidth when the background traffic is a UDP flow. A *SNMP Collector* is deployed to monitor the available bandwidth.

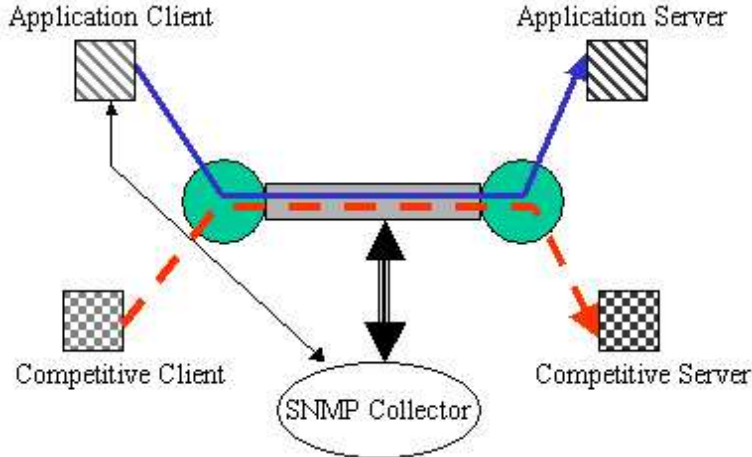
In the experiment, we focus on the comparisons of four pairs of parameters:

1. *predicted\_total\_time* and *total\_time*
2. *predicted\_compr\_time* and *compr\_time*
3. *predicted\_send\_time* and *send\_time*
4. *predicted\_compr\_size* and *compr\_size*

Here, *predicted\_total\_time*, *predicted\_compr\_time*, *predicted\_send\_time* and *predicted\_compr\_size* are the values computed according to formulas (2) - (5). *Total\_time*, *compr\_time*, *send\_time* and *compr\_size* are the corresponding measured values in our experiment.

#### 3.2 Compression Speed and Compression Ratio

*Compression speed* is related to the data format and the machine type. The relationship between application performance and host machine parameters is a research topic that is outside of the scope of this paper. During the experiments, we keep using the same



The four squares are host machines, the slashed squares denote the application client and the application server, and the two grid squares are the competing client and the competing server. The two circles are routers.

Figure 5: Experiment setting

	Compression Speed		Compression Ratio	
	Average (MBps)	Std. Dev.(MBps)	Average	Std. Dev.
TXT	0.84569	0.10470	4.0676	1.2047
PS	0.77839	0.18261	3.8816	2.5839
Binaray	0.65497	0.04605	2.0746	0.2169
PDF	0.87562	0.07125	1.1856	0.03035
JPG	0.83335	0.01509	1.0075	0.0099

Table 1: Compression speed and compression ratio of *gzip* with 16KB compression buffer

machine for all the compressions, and make sure that our application is the only workload. This way, we can think of *compression speed* as a function of compression algorithm. The *compression speed* is also affected by compression buffer size, but we omit this factor by using the same size of buffer, which is 16KB.

The method to get the *compression speed* and *compression ratio* is as follows. Take the same type of data, compress them using *gzip* and record the measured value. By *data type*, we mean the type of data file, for example, binary code file, postscript file, text file, JPEG file, etc. Table 1 presents the empirical data that we get using this method. For each type of data, we randomly download 100 - 110 files from the Internet, and for each data file, repeat the same compression procedure six times. From Table 1, we can see the standard deviation is quite small, which make us believe file type is a reasonable way to differentiate data when considering the general purpose compression algorithm - *gzip*. In our experiment, the source data is a big tar file of a collection of binary files, and we simply use the value from Table 1, indexed

by data type (file type).

### 3.3 Available Bandwidth

In the experiment, the application depends on Remos to provide available bandwidth information. Clearly, the accuracy of the network performance information can affect the final prediction accuracy significantly. We first use a simple experiment to get an idea of the accuracy of the Remos information.

Fig. 6 shows the experimental results. In the experiment, we keep monitoring the available bandwidth of the network path between two machines on our testbed, which generally carries no traffic. The measured value is plotted using the solid line in the figure. At around 6 second (*competing flow starts* in Fig. 6), a 6Mbps UDP flow is added to generate the competing traffic. We can see, with the introduction of competing traffic, Remos starts reporting reduced available bandwidth. After around 4 seconds, the reported value gets stable, with the final value around 3.4Mbps. We think it is reasonable since the link capacity is set as 10Mbps.

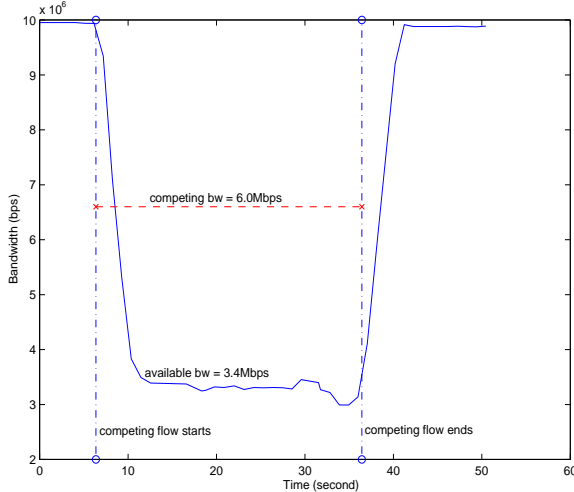


Figure 6: Accuracy of Remos available bandwidth prediction

The reason that we use UDP traffic instead of TCP traffic as the competing flow is that UDP traffic sender side throughput is not affected by other traffic on the same path. In our experiment, we need different rates of competing traffic, so we can easily control its traffic throughput.

### 3.4 Application Performance

To evaluate application performance, we transfer the same amount of data with compression and without compression, recording the execution times.

```

repeat {
    COMPRESS(data, &compr_data);
    SEND(compr_data);
} until (all data is sent out);

```

Figure 7: Pseudocode for data transmission with compression

```

repeat {
    SEND(data);
} until (all data is sent out);

```

Figure 8: Pseudocode for data transmission without compression

In order to know whether the application would make the right decision for using compression, we

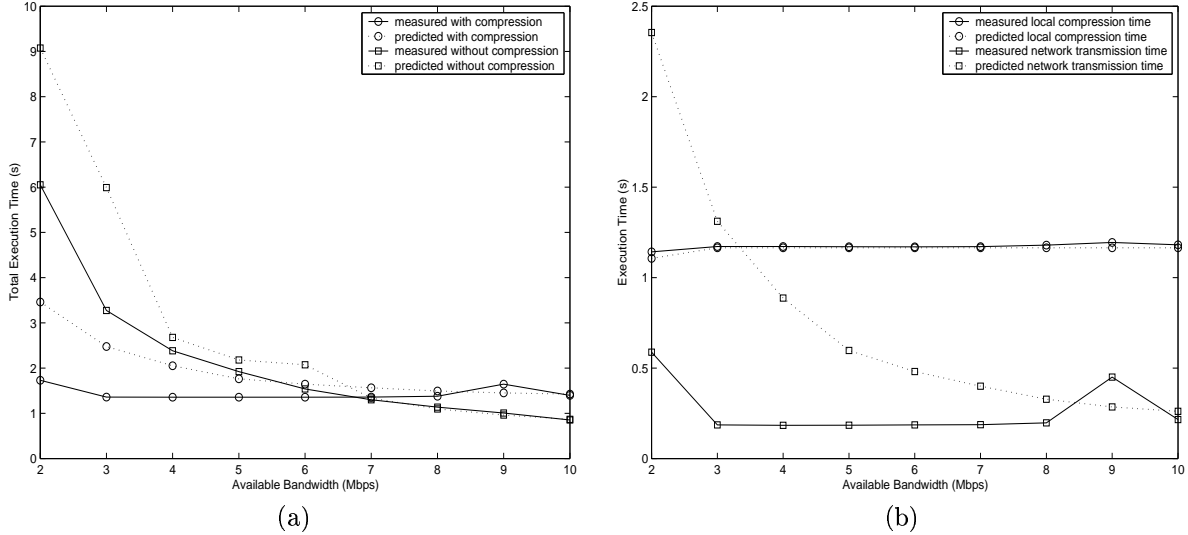
did experiments on data transmission with and without compression separately, measuring and comparing their execution time. The experiment pseudocode is shown in Fig. 7 and Fig. 8; the data is split into multiple chunks, and it process one chunk each loop. We believe this is the most general way of transferring large data set. We repeat the same experiment 20 times with the size of the data involved in network transmission increased by 1MB each loop, where the data size starts from 1MB, and the data source is a large binary code file. We take the averaged value as the final measurement. We repeat the same experiment, with competing traffic changing from 0 to 8Mbps.

Fig. 9(a) shows the predicted and measured total execution time for both in compression and in uncompression mode. Although in some cases, the absolute error in this figure is not minor, the predicted relationship between the elapsed time with and without compression is the same as that of the measured values. When the available bandwidth is less than 7Mbps, compression mode is faster than uncompression mode; and the uncompression mode shows its advantage when the available bandwidth is bigger than 7Mbps. That means that the application in Section 2 would make the correct decision.

Fig. 9(b) shows the application measured local compression time and the network transmission time when we use compression in data transmission. We can see that the local compression time does not show significant changes, which is easy to understand, since it is not affected by the available bandwidth. But the network transmission time looks strange: it does not change very much as the available bandwidth changes from 3Mbps to 10Mbps, unlike the predicted value, which decreases proportionally with the increase of available bandwidth. The reason, which will be discussed in detail in Section 4, is that with the interleaving of local compression and network transmission, there exists execution overlap between the *Compression Module* and the *Network Module*, and the application measured network overhead is actually not the real data transmission time. That basically invalidates the simple model shown in formulas (1) - (5).

### 3.5 Breakdown of Compressed Data Transmission Time

In Fig. 9, each data point is the average value of 20 experiment results. In this section, we illustrate the detail of a single experiment by showing the performance of each module of the application, in order to get an idea where the error comes from.



(a) gives the change of total execution time for data transmission with (circle points) and without (square points) compression, both measured (solid line) and predicted (dot line) values are plotted; (b) is the breakdown the time of data transmission with compression into local compression (circle points) and network transmission (square points)

Figure 9: Application performance

Fig. 10 shows the experimental results for the compressed data transmission, with 6Mbps of UDP competing traffic. Each point in this figure represents one experiment. The same experiment is repeated 52 times, with the transmitted data size increased by 16KB each time. The four figures show the predicted and measured values for *total execution time*, *compression time*, *sending time*, and *data size after compression*. Circle points are predicted value, and star points are measured value. The prediction for compression time is very accurate (see the second and third figure). It shows big difference only in two cases, which can be explained by temporary host system perturbations. Prediction for data transmission time shows a somewhat large difference with the measured value, we will discuss it in Section 3.6. But the overall error is not significant, as shown in the first figure in Fig. 10. This figure shows that the application prediction error mainly comes from the data transmission part.

### 3.6 Analysis

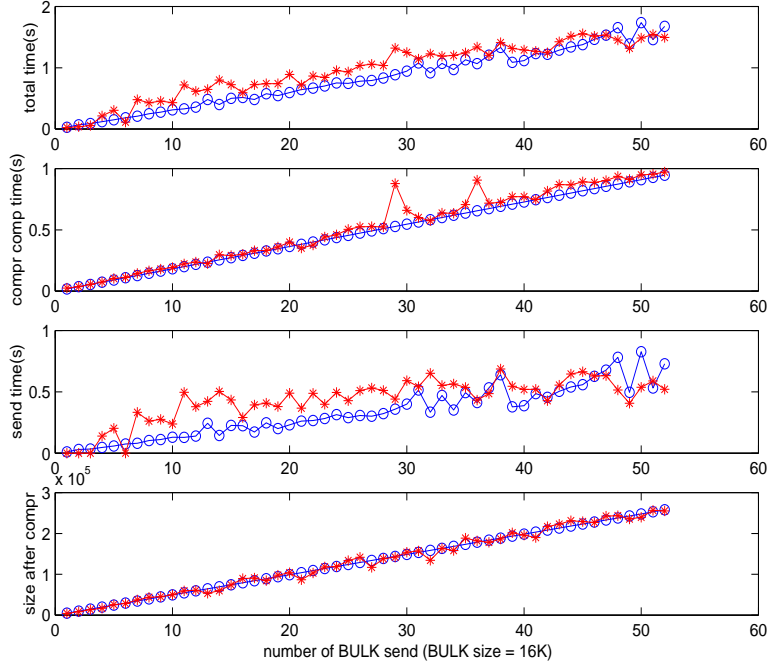
In Fig. 10, we know its error mainly comes from the error of network transmission time. In this section, we systematically evaluate the prediction error of the simple model by making a more complete exploration of the experiment setup. That is, we change the parameters for *Compression Module* and *Network Module*, repeat the same experiment, and measure the

prediction errors.

Fig. 11(a) - (d) shows the errors of the prediction model. In Fig. 11(a) and (c), we change the throughput of UDP competing traffics, from 2Mbps to 8Mbps, while keeping the other parameters constant. Fig. 11(a) is the absolute error and Fig.11(c) is the corresponding relative error for *total execution time*, *compression time* and *sending time*.

The prediction error for compression time is less than 4%, and the errors show no relationship with the available bandwidth, which is easy to understand.

Fig. 11(c) shows that the prediction error for the total execution time mainly comes from that of the sending time prediction. Actually, the predicted values of sending time are generally 2-4 times the measured values (this big absolute error only show a small relative error in Fig. 11(c) is because the data sending time is only small part of the total execution time, around 25%). This is because when the socket API sends out data, it first copies the application data into a kernel buffer, and returns after the copying is finished, regardless of whether the transmission has finished. So when we try to measure the processing time of the socket API call, what we get is actually the data copying time, and not the data transmission time. The application could provide data fast enough to make socket API blocks on the socket buffer. When the available bandwidth is high enough (higher than 3Mbps in Fig.11), the network system can clear the socket buffer fast enough so that the



Circle points are the predicted values, and star points are the measure value. X-axis is the number of data, with transmitted data size increased by 16KB each time. For the Y-axis, the first figure is the total execution time (*total\_time*) computed in formula (2); the second figure is the compression computation time (*compr\_time*) for formula (3); the third figure is the compressed data transmission time (*send\_time*) in formula (4); and the last figure shows the size of the compressed data (*compr\_size*) in formula (5).

Figure 10: Breakdown of compressed data transmission execution time

socket API does not block.

Although when the available bandwidth is less than 4Mbps, the socket API blocks due to the slow network transmission rate, the application will start noticing the changes of available bandwidth, the error of link capacity provided by SNMP agent will start contributing to the prediction error for data transmission. That is, for a link with capacity 10Mbps, the real highest throughput that the application can achieve is actually not exactly 10Mbps, and this error will become more and more significant when reducing the available bandwidth. That is why I see big errors when competing flow bandwidth increases in the second figure of Fig. 11(a).

In Fig. 11(b) and (d), we keep the competing traffic constant, and change the chunk size processed each loop from 4KB to 32KB. The four figures show the average difference between predicted values and measured values together with their standard deviation for *total execution time*, *data transfer time*, *compression time* and *compressed data size*. In Fig. 11(d), the prediction error for compression time is also very small, which is less than 10%. And there is large error for data sending time prediction, the reason is

similar with that of Fig. 11(c). We also notice the prediction errors tend to reduce with the increasing of data size per loop. It is easy to understand since a larger data size per loop will reduce the number of loops to process data, and fewer loop errors will be accumulated.

## 4 Model the Overlap of Local Computation and Network Execution

As mentioned in the previous section, in Fig. 9(b), when the available bandwidth is higher than 3Mbps, the execution time with compression does not change very much. This result does not comply with the prediction formulas (2) - (5), which says that with the decreasing of available bandwidth, transmission time should increase, finally increasing the total execution time. In this section, we show that this abnormal result is due to the overlap between network transmission and local compression on the host. More specifically, if the computation time is big enough, what really matters for the total execution time is the socket

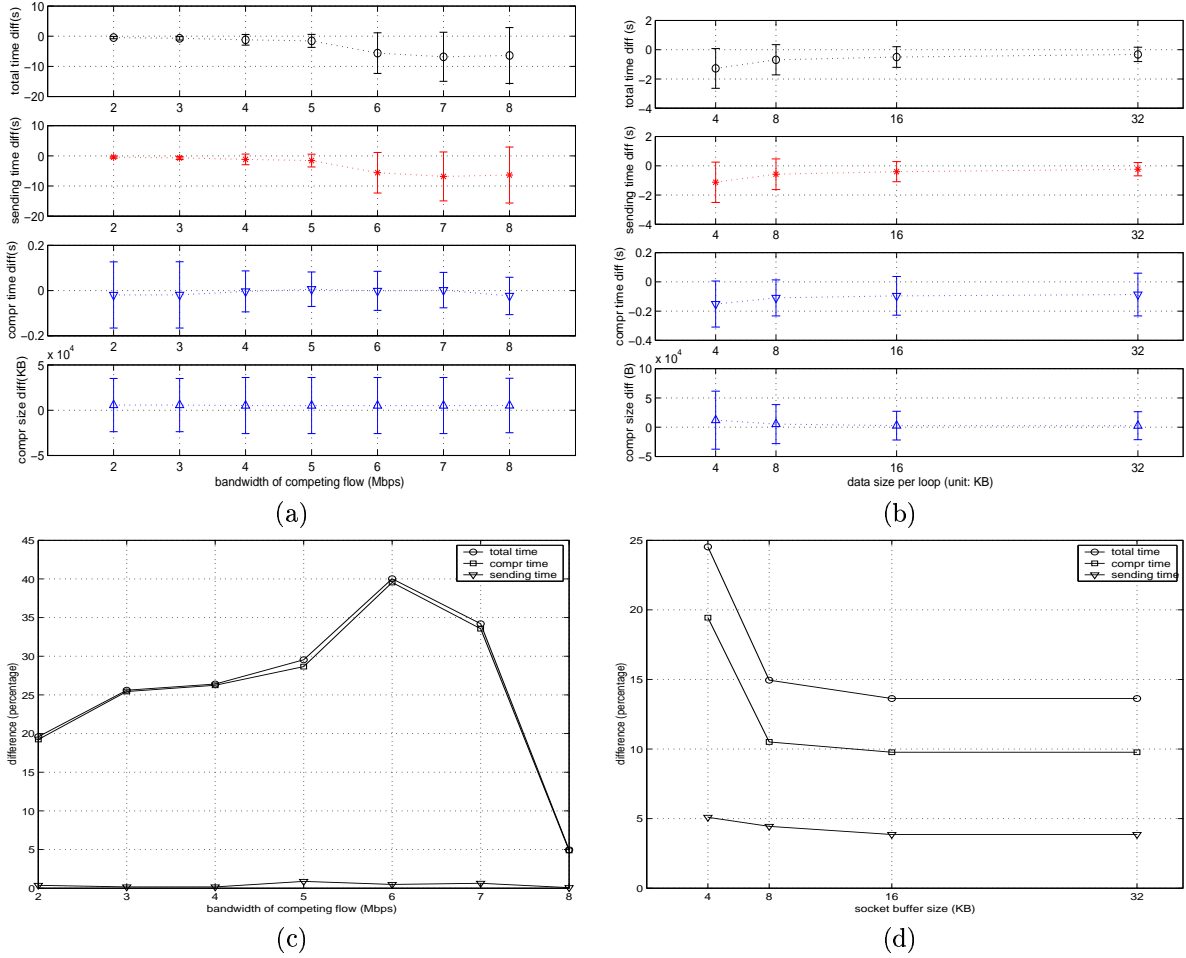


Figure 11: Prediction Error

buffer copying time, not network transmission time. That is why the network available bandwidth will not affect the application performance.

#### 4.1 Theoretical Model

We base our analysis on the execution model in Fig. 7. It is composed of a compression-transmission loop, where each chunk of data is first processed by a computation procedure before it is sent out,

In our implementation, *send()* returns as soon as all the application data has been copied into the kernel buffer. So part of the time used for data transmission will overlap with that of *COMPRESS()*. Assume  $t_c$  is the time of compressing one chunk of data,  $t_b$  is the socket buffer copying time for the compressed data, and  $t_s$  is the corresponding network transmis-

sion time. Let us denote the processing time of all the loops as  $T$ , then this execution model can be expressed as Fig. 12.

Generally,  $t_s$  starts somewhat later than the corresponding  $t_b$ , but the time interval is very difficult to measure. So we simply assume that  $t_s$  and  $t_b$  start at the same time. From Fig. 12,  $T$  can be expressed by the following formula:

$$T = \begin{cases} n \cdot t_c + (n - 1)t_b + t_s & t_c \geq t_s - t_b \\ t_c + n \cdot t_s & t_c < t_s - t_b \end{cases} \quad (6)$$

where  $n$  is the number of loops in Fig. 7. The relationship between compression time and the total processing time in the above formula is illustrated in Fig. 13

It is easy to see, when computation time is big enough ( $t_c \geq t_s - t_b$ ),  $t_s$  is not the dominating factor in the total execution time. Consequently, the

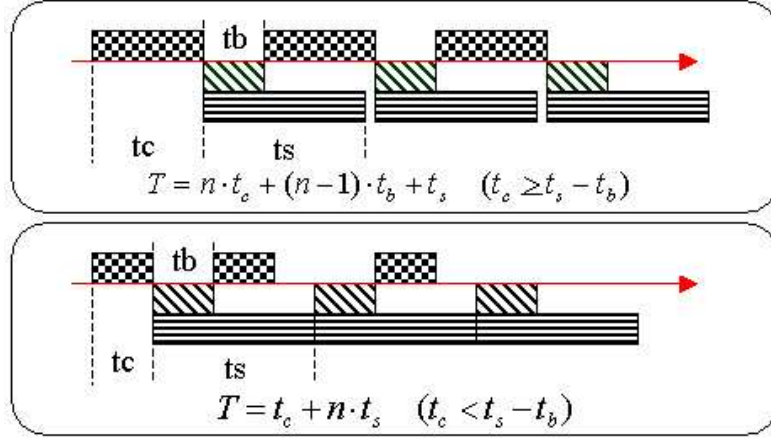


Figure 12: Overlap between data transmission and local compression

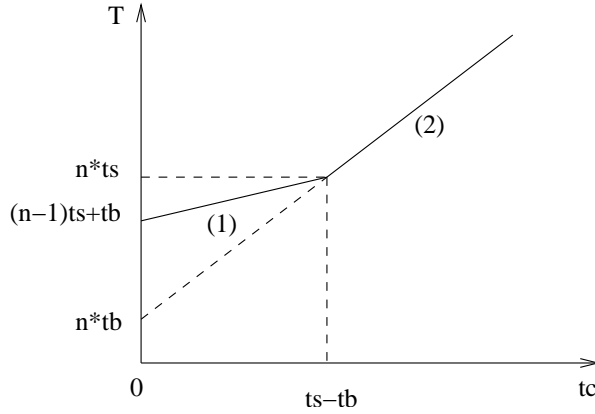


Figure 13: Theoretical model for total execution time, considering the overlap between network processing and local compression

changes of network bandwidth will not affect application performance. The following section shows the experimental result that confirms this claim.

## 4.2 Experimental Setup

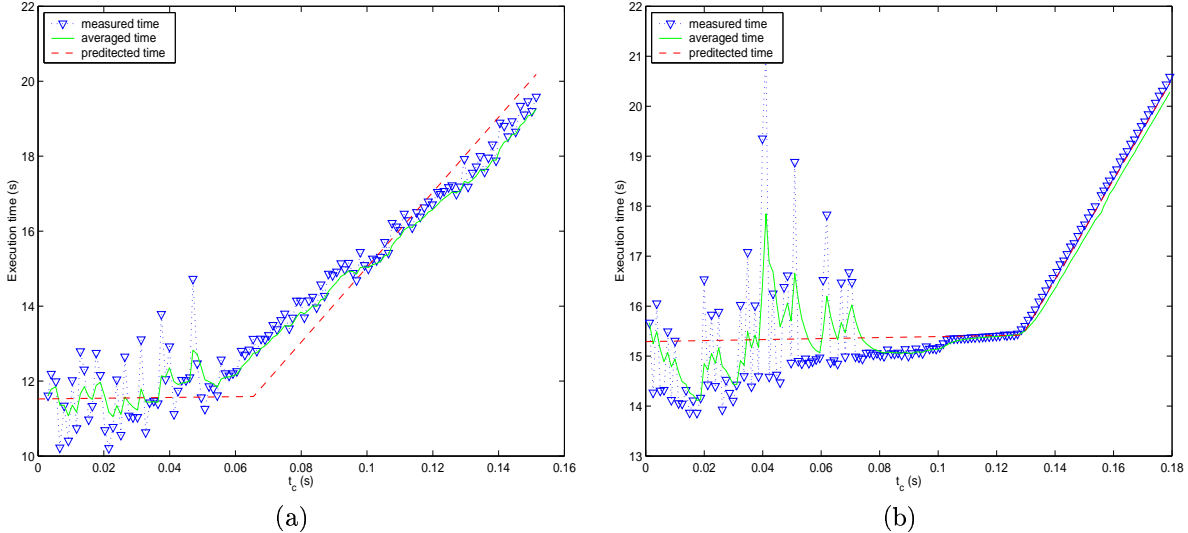
The high level idea of the experiment is the same as that in Fig. 7. We keep running the same compression-transmission loop, measure the execution time of different modules, and try to illustrate the relationship between total execution time and computation time. In our implementation, the computation time should be controllable on a very small time scale, and we should be able to precisely con-

trol the computation time, which is very difficult to do for a compression module. With these implementation requirements, we replace the *compression* part in Fig. 7 with a loop of simple arithmetic calculation. The other part of the loop, *send()*, is implemented by a routine socket data sending procedure.

The experimental setting is the same as that in Fig. 5. The bottleneck link bandwidth is 10Mbps, socket buffer size is 16KB, and each chunk of data is also 16KB. A 6Mbps UDP competing flow is added to change the available bandwidth. For each computation time, we repeat the computation-transmission loop 100 times, recording the average value for computation time  $t_c$  and buffer copying time  $t_b$ .

## 4.3 Experimental Results

Fig. 14(a) and (b) show the experimental results without and with UDP competing flow, respectively. The dashed line is computed using formula (6). In order to calculate the theoretical value for  $T$ , we need to know  $t_c$ ,  $t_b$ , and  $t_s$ . The former two can be measured directly in the experiment.  $t_s$  is chosen manually from the trace data. We think the measured network transmission time as  $t_s$  when  $t_c$  is small enough, because the computation part will be completely covered by network transmission. This method of computing  $t_s$  is not 100% accurate. The error, we think, leads to the difference between measured value (triangle points) and predicted value (dashed line) in Fig. 14(a). Comparing Fig. 14 with Fig. 13, we can see that the experimental result follows our execution model very well, which we believe confirms the



For both figures, X-axis is the computation time  $t_c$  (the corresponding  $t_s$  is the network transmission time for 16KB data), Y-axis is the total execution time for 100 repetition of the computation-transmission loop. Triangle points are the measured value, and dashed line is drawn according to formula (6). (a) shows the data measured without competing traffic, and (b) shows the data measured with 6Mbps UDP competing traffic.

Figure 14: Experiment about improved model

competing BW (Mbps)	64K data transfer time (s)	estimated available BW (Mbps)
0	0.115	4.5
6	0.155	3.4

Table 2: Estimate the Available Bandwidth

improved model in Fig. 12.

Fig. 14 also provides a way to estimate the real data transfer time  $t_s$  (i.e., the *turning point* between segment (1) and (2) in Fig. 13), which can be further used for available bandwidth computation. For example, fitting Fig. 14 into Fig. 13, we can get the data as in Table 2.

Unfortunately, the estimated values does not make much sense. We think it is due to our assumption that socket *send()* ( $t_s$ ) starts sending data at the same time as that of buffer copying ( $t_b$ ), which may not be true in reality. The real data transfer time should be smaller than  $t_s$ , but the exact difference is difficult to measure.

#### 4.4 Analysis

With the results from formula (6) we can give an explanation for the data in Fig. 9. When the available bandwidth is over 3Mbps, the compression time is larger than  $(t_s - t_b)$ . That is why the change of available bandwidth from 4Mbps to 10Mbps does not affect the total execution time, and we see no changes

in the total execution time with the increase of competing flow bandwidth.

With the improved model, we can improve our prediction model in the following way. We can still use formula (1) to compute the processing time in un-compression mode. But for the compression mode, we should use formula (6) instead of formula (2). Given the data size and the performance parameters from *Compression Module* and *Network Module*, it is not difficult to calculate  $t_c$  and  $t_s$ , while  $t_b$  can be measured directly in the application, that is, the execution time that the application sees from socket function *send()* is actually the socket buffer copying time.

## 5 Related Work

Network-aware applications are a hot research area which has been widely discussed. This type of application can be classified by their adaptation behavior. [24] gives a discussion about adaptation models used by network aware application, and [1] presented a framework-based approach to develop net-

work aware applications. Some related work in the mobile computing environment is implemented in Odyssey [5, 16, 20, 21].

Compression before transmission and related pre-processing techniques have been used by many network applications [8, 9, 10, 14, 26]. Among them, [8] and [14] discussed two examples which are very similar with our work. [8] talks about the trade-off between the compression ration and the compression time. It presents a system to automatically and dynamically select the compression format to reduce the *Total Delay* based on the future resource performance. Similar to their work, which uses NWS[25] to detect network performance, we also uses an existing network monitoring system to help predict network performance. But we focus on the judgment whether or not to use compression, not compression format, since compression does not necessarily improve the application's performance.

[14] talks about how to use compression techniques in a transcoding proxy in a mobile network environment. By predicting transcoding delay, transcoding size and network bandwidth, it determines whether to transcode and how much to transcode an image for store-and-forward transcoding and streamed transcoding. The problem they focus on is similar to ours, trying to decide which data to send onto the network, but they uses a different way to monitor and predict the network performance, which is very similar with that of [23].

Neither of these works consider the possibility of overlap among different processing modules. Computing the total execution time as the simple arithmetic summary of each module's execution time, in many cases, can impair application performance.

As a key component of the application, available bandwidth prediction is a hot research topic. IDMaps[6] suggests a scalable Internet-wide architecture, which measures and disseminates distance information on the global Internet. NWS[25] is trying to provide accurate forecasts of dynamically changing performance characteristics for a distributed set of metacomputing resources. SPAND[23] determines network characteristics by making shared, passive measurements from a collection of hosts. NIMI[17] proposes to deal with this problem from the perspective of infrastructure, trying to provide a large-scale, extensible platform for network measurement. Besides these systems, which need complicated configuration, simple tools like *bprobe/cprobe*[2], *nettimer*[12], *pathchar*[11] and *sting*[22] are also available for network performance measurement.

## 6 Conclusion

To adapt to the dynamic change of network performance, applications can use compression to reduce the network transaction time by reducing the size of the data to be transmitted. But compression also increases the local processing time. The trade off between network transmission time and local processing time should be considered to make the right decision whether or not to compress the data.

In this paper, we split the application into three components: *Compression Module*, *Network Module*, and *Compression Decision Module*. We present our method to get the performance information about compression and network bandwidth. Experiments on a LAN testbed shows that our method can work quite well. We can make accurate judgment with our method.

Our experiment also shows that simple aggregation of processing times is not enough to predict the total execution time accurately. We need to consider the overlap between different types of processing. We present a simple model to explain it. Further experiment confirms our model.

## 7 Future Work

Future work for this paper includes studying of the application's behavior on WAN, since the experiment in this paper is only carried out on the LAN testbed.

We shall also investigate techniques to get better estimation and prediction of available network bandwidth. People have already started looking at this problem. [2, 12, 13] mention some models to do this work. But they all have some limitations in terms of implementation. Accurate and realistic tools to estimate available bandwidth need more work. The improved model discussed in this paper (formula (6) and Fig. 13) actually provides a way to estimate the available bandwidth, but we need to solve some difficult problems before making this model practical, as discussed in Section 4.2.

Another important future work is to make more complete usage of the compression algorithm. The compression algorithm used in this paper, *gzip*, is actually a sophisticated compression algorithm, which allows application to change its compression ratio dynamically. It will improve the application's adaptation ability to network services changes.

## 8 Acknowledgements

I would like to thank Prof. Peter Steenkiste for his discussion and encouragement on this work; Nancy Miller and Christopher Lin for their assistance in setting up the experiment; Yang-Hua Chu and the anonymous reviewers for their helpful comments on an earlier draft of this paper.

## References

- [1] J. Bolliger and T. Gross. *A Framework-Based Approach to the Development of Network-Aware Application*, IEEE Transactions on Software Engineering (Special Issue on Mobility and Network-Aware Computing), May 1998, 24(5), pp. 376-390.
- [2] Robert L. Carter and Mark E. Crovella. *Measuring Bottleneck Link Speed in Packet-Switched Networks*, TR-96-006, Boston University Computer Science Department, March 15, 1996.
- [3] Tony DeWitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Jaspal Subhlok, Dean Sutherland. *ReMoS: A Resource Monitoring System for Network-Aware Applications*, Technical Report, CMU-CS-97-194.
- [4] Peter A. Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Dean Sutherland. *The Architecture of the Remos System*, Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, August 2001, California, USA.
- [5] Jason Flinn, Dushyanth Narayanan and M. Satyanarayanan. *Self-Tuned Remote Execution for Pervasive Computing*, In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII) May 2001, Schloss Elmau, 82493 Elmau/Oberbayern, Germany.
- [6] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, L. Zhang. *IDMaps: A Global Internet Host Distance Estimation Service*, To appear in IEEE/ACM Trans. on Networking, Oct. 2001.
- [7] GZIP. <http://www.gzip.org/>.
- [8] Richard Han, Pravin Bhagwat, Richard LaMaire, Todd Mummert, Veronique Perret, and Jim Rubas. *Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing*, IEEE Personal Communications Magazine, December 1998.
- [9] Hemy, M., Hengartner, U., Steenkiste, P., and Gross, T.. *MPEG System Streams in Best-Effort Networks*, Proc. of PacketVideo '99, New York, April 1999.
- [10] Michael Hemy, Peter Steenkiste, and Thomas Gross. *Evaluation of Adaptive Filtering of MPEG System Streams in IP Networks*, IEEE International Conference on Multimedia and Expo 2000, New York, New York.
- [11] Van Jacobson. *pathchar - a tool to infer characteristics of Internet paths*, presented as April 97 MSRI talk.
- [12] Kevin Lai and Mary Baker. *Nettimer: A Tool for Measuring Bottleneck Link Bandwidth*, Proceedings of the USENIX Symposium on Internet Technologies and Systems, March 2001.
- [13] M. Kim and B. D. Noble. *Mobile network estimation*, in the Seventh ACM Conference on Mobile Computing and Networking, July 2001, Rome, Italy.
- [14] C. Krintz and B. Calder. *Reducing Delay with Dynamic Selection of Compression Formats*, Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, August 2001, California, USA.
- [15] Nancy Miller and Peter Steenkiste. *Collecting Network Status Information for Network-Aware Applications*, Proceedings of INFOCOM 2000.
- [16] Brian D. Noble. *Mobile Data Access*, Ph.D. Thesis, CMU-CS-98-118, May 11, 1998.
- [17] Vern Paxson, Andrew Adams and Matt Mathis. *Experiences with NIMI*, In Proceedings of the Passive and Active Measurement Workshop, 2000.
- [18] RFC 1157. *A Simple Network Management Protocol (SNMP)*.
- [19] RFC 1213. *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*.
- [20] M. Satyanarayanan. *Pervasive Computing: Vision and Challenges*, In IEEE Personal Communications, pp. 10-17, August, 2001.
- [21] M. Satyanarayanan. *Mobile Information Access*, In IEEE Personal Communications, Vol. 3, No. 1, February 1996.

- [22] Stefan Savage. *Sting: a TCP-based Network Measurement Tool*, Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems, pp. 71-79, Boulder, CO, October 1999.
- [23] S. Seshan, M. Stemm, R. H. Katz. *SPAND: shared Passive Network Performance Discovery*, In Proc 1st Usenix Symposium on Internet Technologies and Systems (USITS '97) Monterey, CA December 1997.
- [24] Peter Steenkiste. *Adaptation Models for Network-Aware Distributed Computations* 3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99), Orlando, January 8, 1999.
- [25] Rich Wolski, Neil T. Spring, and Jim Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, Journal of Future Generation Computing Systems, 1999, also UCSD Technical Report Number TR-CS98-599, September, 1998.
- [26] N. Yeadon, F. Garcia, D. Hutchison, and D. Shepherd. *Filters: Qos support Mechanisms for Multipeer Communications*, IEEE Journal on Selected Areas in Communications, 14(7):1245-1262, Sept 1996.