

Bandwidth Monitoring for Network-Aware Applications

Jürg Bolliger and Thomas R. Gross

Laboratory for Software Technology
Departement Informatik
ETH Zürich
CH 8092 Zürich, Switzerland

Abstract

Network-aware content delivery is an attractive approach to mitigate the problems produced by the significant fluctuations of the available bandwidth present in today's Internet. Such network-aware applications require information about the current condition of the network to adapt their resource demands. Such information can be obtained at the network level, the transport protocol level, or directly by the application. This paper compares two kinds of application-level monitoring (at the sender and receiver side) and transport-level monitoring with regard to their ability to provide useful information to network-aware applications. Our results indicate that transport-level monitoring can effectively and efficiently satisfy the need for timely and accurate bandwidth information. This observation has direct implications for protocol/OS design: It is desirable to widen the transport layer API to allow the application efficient access to network status information required for adaptation.

1 Introduction

Current best-effort networks create a challenge for application developers that want to provide predictable performance to users. The effective performance realized in a network that consists of many heterogeneous links and that is subject to congestion can vary dramatically over time and space. Applications that aim to deliver a response in a fixed time interval are in a difficult position.

Recently, a number of researchers have proposed *network-awareness* as a mechanism to bridge the wide performance gaps and to cope with the significant bandwidth volatility that may be encountered. A network-aware application adapts its content delivery in response to network conditions so that the application's demands do not exceed the bandwidth available (to this application). In

times of network resource shortage, objects are dynamically transcoded to reduce the amount of data that must be transmitted. A number of network-aware applications have been implemented and provide evidence that network-aware content delivery is capable of adapting to the varying supply of bandwidth.

Since network-aware applications can be deployed in the current network infrastructure (without requiring the introduction of new service models like differentiated services [1] or reservations [17]), they are an attractive way to deal with the bandwidth heterogeneity and fluctuations observed in current networks. To adapt to changing network conditions, an application needs information about the current network status. In this paper we investigate different options on *how* to gather network status information. Our focus is on providing information to a single application even in the absence of a network infrastructure like Remos [8, 6].

Gathering data is the first step in a chain that allows an application to adjust its behavior. Other important issues are how an application can use measurements of the current network status to extrapolate into the future and how to model and select appropriate adaptation strategies. These topics are beyond the scope of this paper – here we concentrate on the basic question of where to collect measurements. The nature of our investigation requires that we use extensive simulation. Wherever possible we have used traces (or constraints) from a real-life network-aware image server [14] to guide our investigation. The user can control the transfer of the images by specifying a limit on the response time. The network-aware image server tries to deliver the relevant images on time by adapting the quality (size) of the images, if necessary. Trading quality for response time, the server tries to maximize the quality of the images delivered within the time frame allotted by the user. At the core of the server is a feedback loop that adjusts the size of each object; for each object, a separate transcoding decision is possible. The model and the implementation assume that transcoding in-

curs a latency, which depends on object properties, and that transmission and transcoding of objects can proceed in parallel.

2 Techniques for bandwidth monitoring

There exist three basic approaches to collect information that can be used to estimate the bandwidth that is (or will be) available to an application:

Application-level monitoring. Obviously, the application is in a good position to monitor the bandwidth it gets. Monitoring can be done either at the sender or at the receiver: e.g., a sender can monitor how fast it can pump data into the network¹, and a receiver can monitor at which rate the data is delivered by the network.

Transport-level monitoring. Since congestion-controlled transport protocols gather a number of performance metrics to adapt the transmission rate of a connection such that it matches the current congestion state in the network, they have most of the information needed by a network-aware application readily available. E.g., for TCP, the current congestion window and round-trip time are readily available. Other metrics such as loss rate, number and duration of timeouts, etc. can easily be determined. Recently, models have been proposed that use such transport-level information to estimate the bandwidth of a single TCP connection [10, 13, 5]. However, this information is currently not available to an application.

Network-level monitoring. A wealth of useful information can be collected at the network level by tools such as Remos [8, 11, 9]. Such approaches can also reveal useful information beyond the current bandwidth, e.g., topology information.

Of these three approaches, we do not pursue network-level monitoring further in the context of this paper. We want to investigate how network-aware applications can obtain information even in the absence of an infrastructure like Remos, because there are many environments (e.g., WANs, Internet) where such infrastructure support cannot be counted on.

The following sections answer the questions: How do these monitoring techniques compare in terms of efficiency and quality (accuracy and timeliness) of the bandwidth information? How difficult it is to implement a transport-level monitor?

¹With a congestion-aware transport protocol, this rate is constrained by flow and congestion control.

2.1 Monitor architecture

To allow for an unbiased comparison of the overheads incurred by the different approaches to information collection, these approaches must be integrated in a single monitoring architecture. An integrated architecture ensures that our conclusions about the efficiency of the different monitoring techniques are not disturbed by differences in the implementation. Furthermore, to compare the different approaches with respect to the accuracy of the bandwidth information, the system must allow a connection to be monitored with multiple techniques simultaneously.

These requirements lead us to pursue a two-tier approach for our monitor architecture. *Sensors* are responsible for the collection of raw network status and performance samples. *Observer* components are responsible for application-oriented functionality, e.g., for aggregating performance information of connections sharing a network path, or forming bandwidth estimates from the raw performance samples. Figure 1 depicts an overview of the components of the monitor architecture.

An observer component is realized as a daemon (*observed*) that can be run on any host in the network. The observer manages and caches the state and performance information of multiple active connections simultaneously. The daemon accepts and processes various types of messages, e.g., messages that indicate the “arrival” or the “departure” of a connection (*register* and *unregister*), and messages that deliver performance *updates* from the sensors. The observer can be queried both for bandwidth estimates for individual connections, or for estimates on the aggregate bandwidth of multiple connections between two hosts (*getinfo/info*). Multiple applications may share an observer component.

Sensors are conceptually simple components that collect raw performance data, encapsulate the data in update messages, and send the messages to the observer that processes the performance updates. There are different types of sensors that collect raw, network-oriented performance data—one sensor for each alternative of information collection. In Figure 1, the different types of sensors are highlighted with different degrees of shading. The sensors communicate with *observed* by means of UDP messages.

2.2 Information collection

Application-level, sender-based information collection is performed by the sensor denoted as *app-snd*. Application-level, receiver-based information collection is done by the *app-rcv* sensor. Information collection at the application-level is achieved by wrapping the send and receive functions of the socket API. These wrapper functions update the total count of bytes sent or received and record a timestamp. If the time passed since the last update mes-

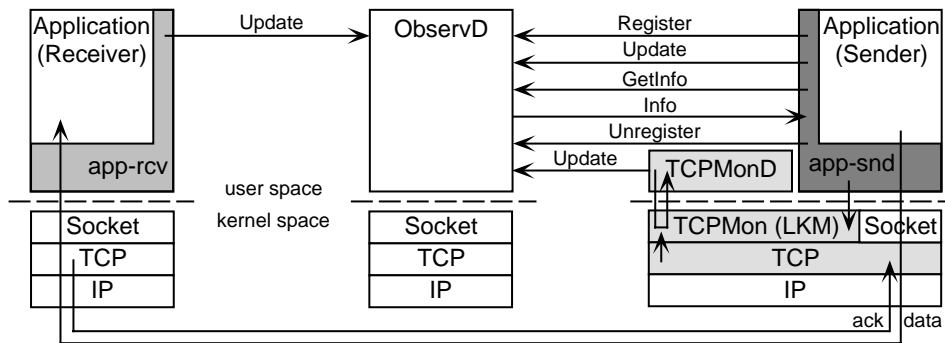


Figure 1. Simple monitor architecture that integrates different approaches to information collection.

sage has been sent to the observer exceeds $1/f$, where f is the sampling frequency, then a new performance report is generated.

Information collection for transport-level monitoring is implemented by a layered architecture and must be collocated with the sending application. The three layers of the approach (see Figure 1) include a few hooks in the TCP stack, a loadable kernel module (LKM) termed *tcpmon*, and a user-level daemon process (*tcpmond*). The rationale behind this layered construction is that we want to introduce as little changes to the TCP stack as possible² and still get access to all the information needed to model the bandwidth available to a TCP connection. We implemented our prototype in NetBSD 1.3, which has a standard and well-documented TCP stack [16].

A few hooks (call-backs) are installed at appropriate places in the TCP stack, so that asynchronous events such as timeouts, etc. can be recorded³. When the LKM is loaded, the hooks are installed and TCP calls back the kernel module to update state information held there. Otherwise, TCP operates as if unchanged.

The LKM implements the call-back functions, manages the association between TCP control blocks and the performance data for the TCP connections, and exports *system calls* to allow sending applications to register and unregister TCP connections to be monitored and to query the status of *all* the monitored connections (*getinfo()*). *getinfo()* returns a list of identifiers (of monitored connections) and a list of performance samples (one per connection).

tcpmond uses this system call to periodically obtain up-to-date status information. *tcpmond* then encapsulates the performance data in an update message which is dispatched to the observer where the TCP throughput models are applied to estimate the bandwidth available to each of these

²In particular, changes to kernel data structures (TCP control block) are avoided completely.

³The information required to model the throughput of a TCP connection is [13, 5]: round-trip time, loss rate, the average duration of a timeout, the receiver's strategy to generate acknowledgments, and the receiver window size.

connections. Note, with this setup, the performance samples of multiple connections are delivered to *observd* in a "batched" manner.

The changes to the TCP stack required to make transport-level monitoring work are very small. In total, the NetBSD 1.3 TCP stack comprises 13 files and a total of 5565 lines of code. Only 50 lines of code in 4 files had to be added to install the call-backs. The LKM is also a small piece of software of 850 lines of code [2].

3 Comparison of monitoring techniques

This section compares the monitoring approaches with respect to the efficiency of the information collection process, and the quality (accuracy and timeliness) of the bandwidth information.

Bandwidth monitoring should impose as little overhead on the network and the end-system(s) as possible. We expect monitoring overhead to be dependent on the (targeted) sampling frequency, and the number of connections that must simultaneously be monitored. Timeliness depends on the frequency and regularity with which performance information is obtained. The (effective) sampling frequency is limited by the overheads incurred by monitoring.

The issue of *accuracy* has been addressed in earlier work [3, 13, 5]. [3, 2] show that transport-level monitoring can accurately model the bandwidth available to an application (the model error is smaller than 20% for 75% of the connections traced in a 6-month Internet experiment [4, 3], and smaller than 50% for 99% of the connections). Moreover, the TCP models [13, 5] capture the (intrinsic) behavior of long-running connections and are able to model the available bandwidth even early in a connection. Therefore, these models may better suit the needs of bandwidth prediction than application-level bandwidth estimates that exhibit larger fluctuations.

3.1 Evaluation methodology

Since both efficiency and timeliness seem to depend on how well the monitor copes with load, we conduct a simple experiment to study the different approaches under varying levels of load, where load is determined by the number of connections n that are monitored simultaneously and the frequency f with which performance reports are generated.

We run the following experiment. A 200 MHz Pentium Pro PC running NetBSD 1.3 (with our TCP modifications) acts as the server machine. A dual-processor 300 MHz SPARC Ultra 4 system running SunOS 5.6 serves as the client machine. The two hosts are connected with a 100 Mbit/s switched Ethernet. *observd* is co-located with the server. The client machine starts n client processes simultaneously. Each client process connects to the server and requests transmission of $100/n$ MB. The client specifies the method for monitoring (*app-rcv*, *app-snd*, or *tcpmon*) and the sampling frequency f , which determines how often the sensor of choice must generate performance reports.

In our experiments, we vary the sampling interval $\delta t = 1/f$ between 0.05 and 2.0 seconds, and choose the number of parallel connections n from the set $\{5, 10, 20, 40\}$. We run 10 experiments for each combination of n , δt , and method of information collection.

3.2 Efficiency

Since the number of performance samples that must be collected and processed each second is equal for all the monitoring techniques and is given by $n \cdot f$, we compare the techniques' efficiency based on the overhead incurred by a single performance sample.

Figures 2 (a)–(d) plot the average overhead incurred (at the observer) by a sample as a function of the sampling interval δt for different numbers n of connections. Two types of costs are reported, the total per-sample costs of information collection (solid lines) and the processing costs for a sample (dashed lines). The processing costs are included in the total costs. The costs reflect the user and system CPU time consumed. The total per-sample costs are obtained by dividing the CPU consumption of *observd* for the whole experiment by the number of all the performance samples that are processed. The per-sample processing costs are measured for each performance sample individually and cover the time needed to process and log the information conveyed in the update message.

The most important aspect to note in Figures 2 (a)–(d) is that *tcpmon* incurs considerably lower total costs per sample than the application-level approaches. This discrepancy can mainly be attributed to the fact that *tcpmon* incurs considerably smaller communication overheads: the performance samples of all n connections are “batched” in *tcpmon*. That

is, the performance samples of all n connections are aggregated and communicated to *observd* in a single update message. This behavior is in contrast to *app-snd* and *app-rcv* where each performance report must be processed individually (because they are generated by different applications). The effect of this “batching” is reflected by the decrease in the total per sample costs as the number of connections increases. This decrease implies that the constant cost of generating and communicating such a batched performance sample is amortized over larger numbers of connections. This argument also explains why the application-level approaches exhibit significantly larger differences between total and processing costs than *tcpmon*.

Figures 3 (a)–(c) present the same data organized by monitoring technique. In addition, these figures show error bars that indicate the confidence interval for the mean at a confidence level of 95%. *tcpmon* scales significantly better than *app-snd* and *app-rcv* as the number of connections is increased.

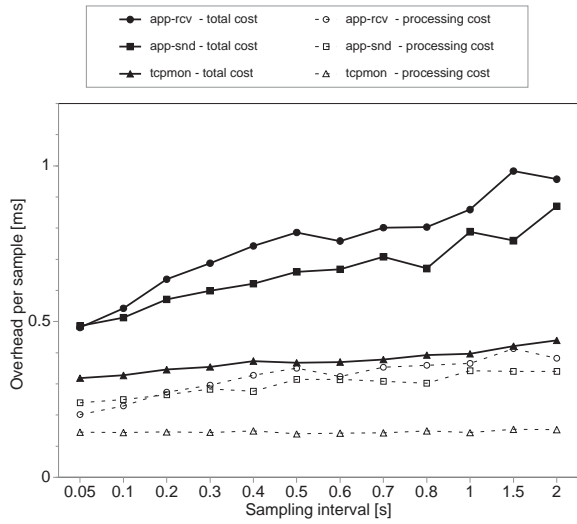
The results are summarized in the first two columns of Table 1. The averages (μ) reported are taken over all the experiments conducted, i.e., over all combinations of n and δt . The confidence interval (at a confidence level of 95%) is $[\mu - e_{95}, \mu + e_{95}]$.

To assess the overhead of information collection in the kernel we experiment with a modified version of *observd* that includes *tcpmond*. In this version, the performance updates are communicated from *tcpmond* to *observd* by means of procedure calls instead of UDP messages. We find that the difference between the total costs (174 μ s) and processing costs (135 μ s) per sample is very small, which indicates that information collection in the TCP stack and the LKM is efficient and incurs small overheads ($\approx 39\mu$ s per sample and connection).

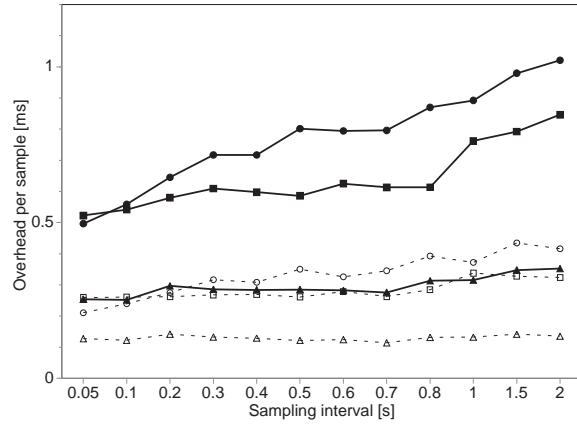
Another efficiency aspect worth discussing is that application-level, receiver-based monitoring incurs considerable network overhead ($n \cdot f \cdot size(update) \approx 25 \cdot n \cdot f$ bytes per second), whereas the other approaches (*app-snd* and *tcpmon*) incur no network overhead.

3.3 Timeliness

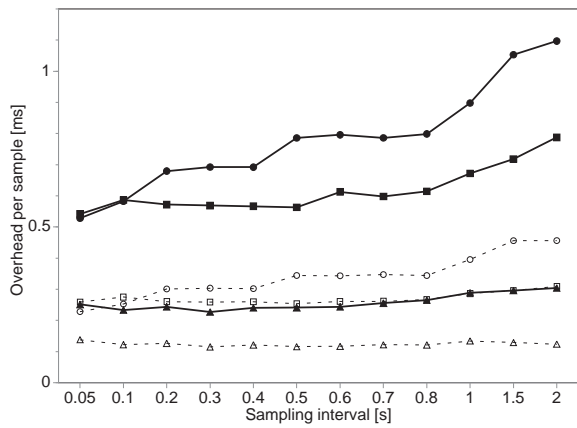
Bandwidth estimates must be provided in a timely fashion, i.e., a bandwidth monitor must detect and report changes in available bandwidth quickly. Timeliness primarily depends on the frequency with which performance information is obtained. The (effective) sampling frequency is limited by the monitoring overheads (see above). Another important aspect of timeliness is *when* the samples arrive, i.e., whether they are regularly spaced or not. If k performance samples arrive per second and all the k performance samples arrive in a time frame $\delta t \ll 1$ second, then all samples except for the most recent sample are useless for the



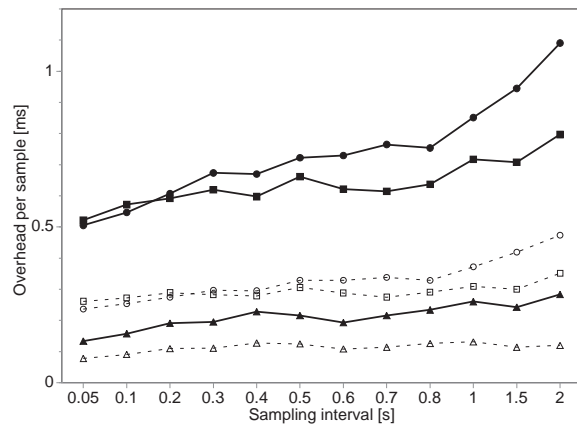
(a) 5 Connections



(b) 10 Connections

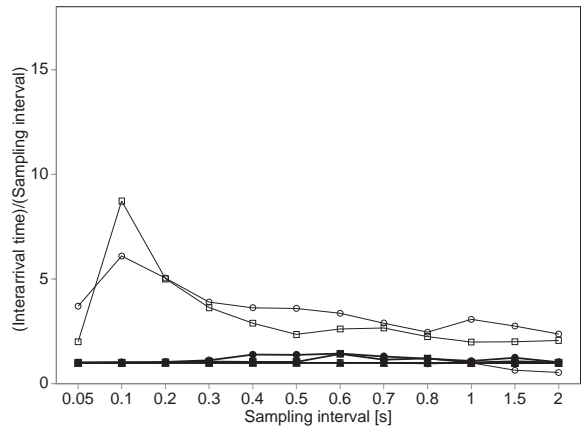
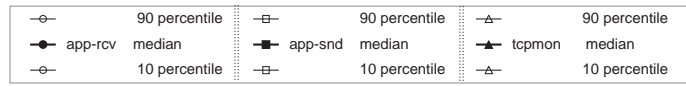


(c) 20 Connections

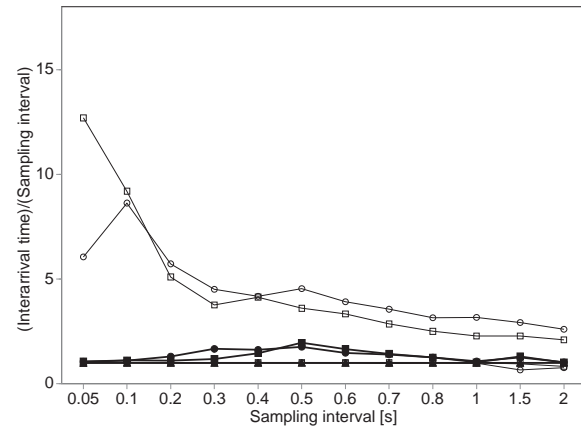


(d) 40 Connections

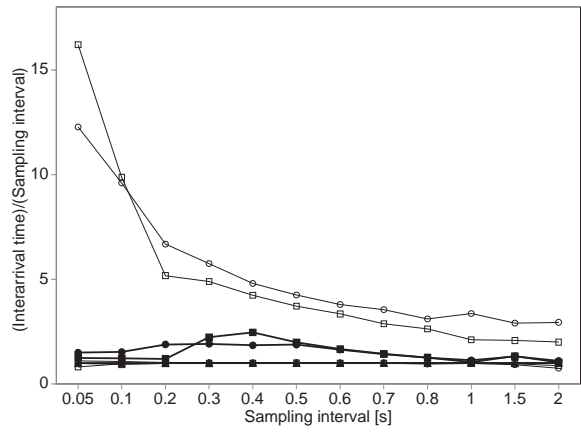
Figure 2. Cost per sample.



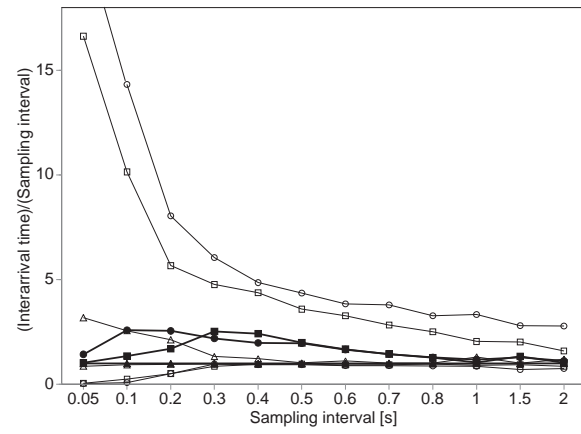
(a) 5 Connections



(b) 10 Connections

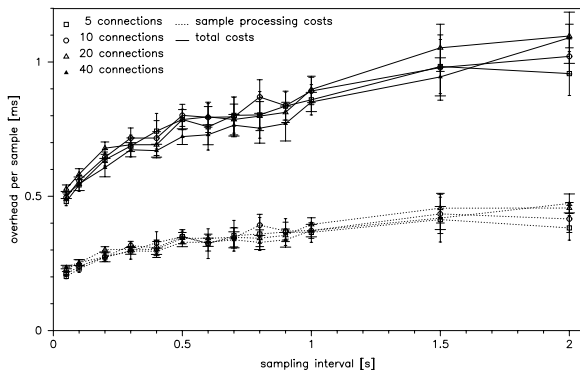


(c) 20 Connections

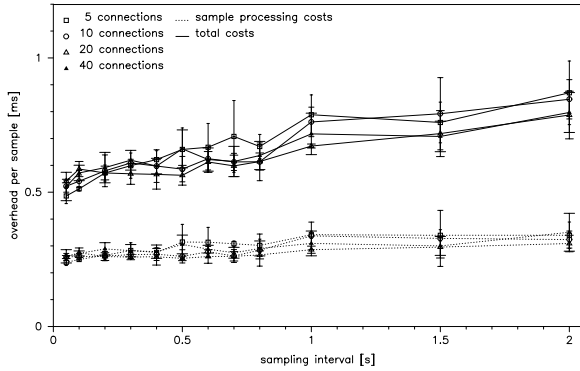


(d) 40 Connections

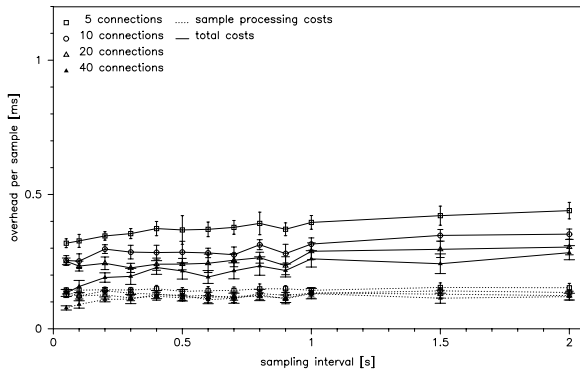
Figure 4. Timeliness of the bandwidth information.



(a) *app-rcv*



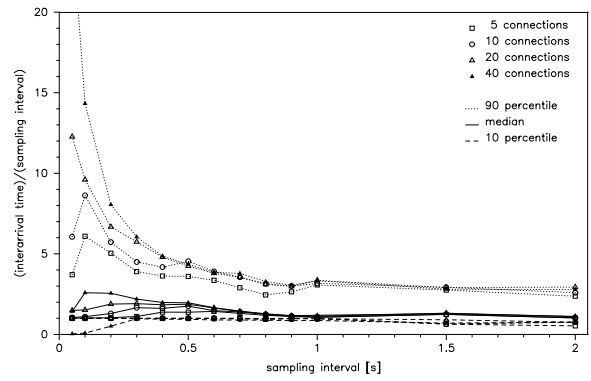
(b) *app-snd*



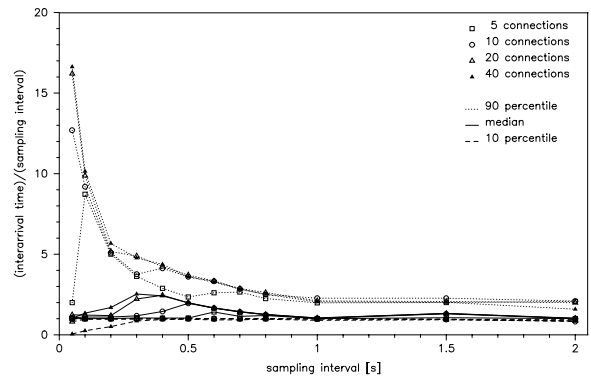
(c) *tcpmon*

Figure 3. Cost per sample, organized by monitoring technique.

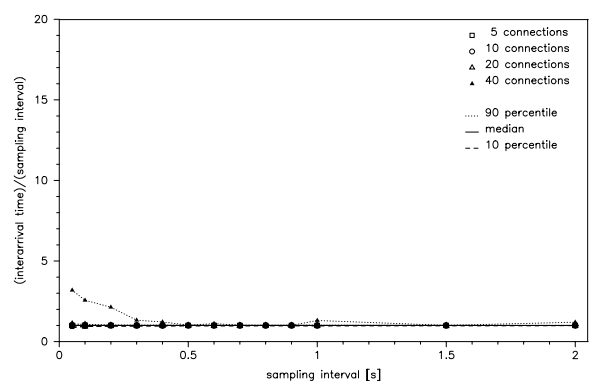
application and simply waste end-system and network resources. We use the interarrival time between successive performance samples for a connection to describe the reg-



(a) *app-rcv*



(b) *app-snd*



(c) *tcpmon*

Figure 5. Timeliness of the bandwidth information, organized by monitoring technique.

ularity of sample arrivals at the observer. We compute the distribution of interarrival times for an experiment and report the 10-, 50-, and 90-percentile divided by the (targeted)

sampling interval δt . A ratio of 1 means that the samples are spaced exactly with the targeted sampling interval δt .

Figures 4 (a)–(d) plot these ratios as a function of δt and n for different a number of connections. We find that application-level monitoring witnesses considerable deviations from the targeted sampling intervals. The median interarrival time is up to a factor three larger than intended for large numbers of connections and high sampling frequencies. Furthermore, the application-level approaches experience considerable variation in the interarrival times as can be seen from the 10- and 90-percentiles⁴. In contrast, transport-level monitoring very accurately matches the targeted sampling intervals for all combinations of n and f . Furthermore, the variability is negligibly small. And as the number of connections is increased, *app-snd* and *app-rcv* exhibit an increase in variability. The 90-percentile for these monitoring techniques increases significantly.

To allow easy comparison with Figure 3, Figures 4 (a)–(d) depicts these data organized by monitoring technique. These figures emphasize the property of *tcpmon* to produce evenly spaced measurement samples. The findings about timeliness are summarized in the last two columns of Table 1.

3.4 Discussion

To summarize, we have shown that the effectiveness of network-aware applications depends considerably on the timeliness and accuracy of bandwidth information. We have demonstrated that the need for accurate and timely information about network resource availability can both effectively and efficiently be satisfied with transport-level monitoring. Furthermore, transport-level monitoring compares favorably with application-level monitoring as far as the timeliness of the bandwidth estimates and the overhead incurred by monitoring are concerned.

Our experience with a prototype monitoring system demonstrates that the implementation of a transport-level monitor requires only minimal changes to existing protocol stacks and should be simple to incorporate in the design of new transport protocols. Our findings also imply that a simple widening of the protocol API suffices to provide network-aware applications with the information about network status sought.

⁴Note that application-level monitoring adapts the delivery of update messages to the transmission rate of application data. This observation in part explains the large variation in interarrival times experienced. Even if we use strict feedback generation (at the targeted sampling frequency) for application-level monitoring, we see smaller, but still significantly higher variation than with transport-level monitoring. However, the overheads of such a strict feedback generation scheme are considerably higher [2].

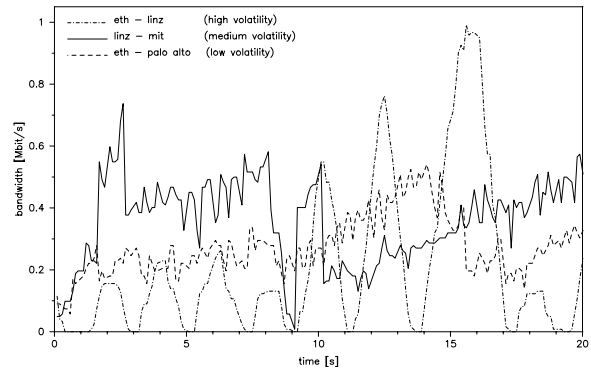


Figure 6. Bandwidth traces of three Internet transfers exhibiting considerable volatility.

4 Accurate and timely feedback

How sensitive is model-based adaptation to the accuracy of the bandwidth information? The effectiveness of adaptation presumably depends on two factors: the delay of the feedback loop and the accuracy of the bandwidth information. The shorter the delay, the better the application will be able to track changes in the network. We suspect that inaccurate information can degrade performance by having the application adapt needlessly, and by having the application select an incorrect operating point.

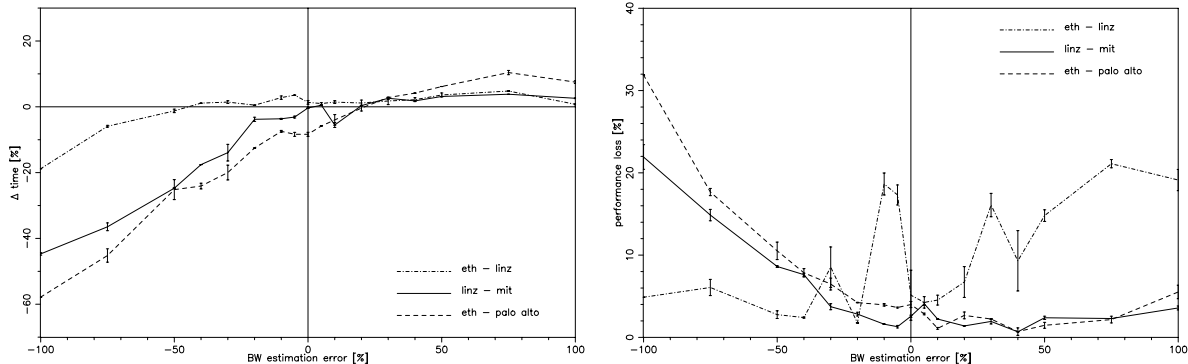
4.1 Accuracy

To study the sensitivity of network-aware delivery to the accuracy of the bandwidth estimation in realistic scenarios (exhibiting varying degrees of bandwidth volatility), we conduct the following experiment. We pick three traces with different levels of volatility out of a set of 4148 TCP connections obtained from a large-scale Internet experiment [4, 3]. Figure 6 shows the three bandwidth traces used for the experiment. The trace “ETH–Linz” represents a high, “Linz–MIT” a medium and “ETH–Palo Alto” a low volatility scenario. Although the bandwidth traces exhibit a wide range of volatility, the connections achieved approximately the same average bandwidth. We use a request for which the image server returns 25 JPEG images totaling 1.03 MB and set the time limit to 15 seconds. We vary the degree of error e introduced in the bandwidth estimates, where $e = 100 \cdot (bw_{estimated} - bw_{actual}) / bw_{actual}$. Each experiment is repeated 5 times.

Figure 7 plots the average relative deviation from the time limit (a) and the average performance loss (b) as a function of the error e introduced in the bandwidth estimates. The error bars depict the confidence interval for the mean at a 95% confidence level. In the cases with low and medium bandwidth volatility, we find that under-estimation

	Cost [μ s]				Timeliness (<i>sample interarrival time/sampling interval</i>)		
	Processing		Total		median	Variability	
	μ	e_{95}	μ	e_{95}		10%	90%
<i>app-rcv</i>	330	7	759	16	1.4	0.9	5.1
<i>app-snd</i>	286	5	639	13	1.4	0.9	4.4
<i>tcpmon</i>	128	2	284	7	1.0	1.0	1.1

Table 1. Comparison of monitoring techniques: summary of results.



(a) Relative deviation from time limit (in %)

(b) Performance loss (= 100% – bandwidth utilization).

Figure 7. Application performance as a function of inaccuracy of bandwidth estimation.

of the bandwidth results in the transfers finishing considerably ahead of time. Consequently, bandwidth utilization suffers badly due to overly conservative adaptation decisions. On the other hand, over-estimation does not seem to have a negative impact, neither timewise nor in terms of bandwidth utilization. To the contrary, it may even lead to improved performance compared to experiments conducted with $e = 0\%$. However, caution must be applied when interpreting these results. If the time limit is shorter and/or the number of images requested is smaller, then the flexibility to react to the effects of consistent over-estimation decreases rapidly and so does performance.

Considering the experiments with high bandwidth volatility we see quite different effects, in particular as far as bandwidth utilization is concerned. The curve named “ETH–Linz” in Figure 7 (b) indicates that inaccurate bandwidth estimates are problematic in situations with highly fluctuating bandwidths. Bandwidth utilization is quite sensitive to even small changes in the accuracy of the estimates. The error bars are wider than for the less volatile bandwidth traces. Although no clear trend is discernible how utilization develops with increasing inaccuracy of the bandwidth estimates, it is important to note that performance problems are likely to arise if the bandwidth estimator performs badly.

In summary, we find that the effects of inaccurate bandwidth estimates depend on the volatility of the bandwidth available. Network-aware delivery is not overly sensitive to the accuracy of bandwidth estimates (in particular for low

and medium levels of bandwidth volatility). However, significant performance penalties may have to be witnessed if bandwidth estimates are inaccurate. This problem seems to be exacerbated in situations with highly fluctuating bandwidths. As a consequence, network-aware applications can benefit considerably from accurate bandwidth estimators. How accurate would such an estimator have to be? The results from the experiments suggest that an accuracy of $\pm 10 - 20\%$ should suffice to stay within $\pm 10\%$ of the time limit.

4.2 Timeliness

To study the effects of untimely (delayed) feedback we conduct experiments similar to the experiments above. We introduce varying degrees of delay into the feedback loop. The details of the experiments and the results have been reported in earlier work [3]. We briefly reiterate the fundamental results because they further support the findings in Section 4.1. The results indicate that the model-based adaptation is not extremely sensitive to delays in the feedback loop and that delayed bandwidth information does *not necessarily* result in lower performance. However, the results also demonstrate that considerable performance penalties *may* have to be witnessed as the bandwidth estimates become less timely.

4.3 Discussion

The performance problems that inaccurate and untimely bandwidth estimates can cause are aggravated if bandwidth fluctuates heavily. As a result, we conclude that timely and accurate bandwidth estimation is important to the efficacy of network-aware applications. This paper discusses the questions of timeliness and accuracy of the feedback signals only in the context of the models for network resource availability. However in many application scenarios, other aspects, e.g., host load, must be considered as well, and similar considerations about accuracy and timeliness apply.

5 Related work

Network resource information that is obtained by a monitor can subsequently be used in a prediction system like NWS (the Network Weather Service) [15]. But we are aware of only three aspects of previous work that discuss the structure of resource monitors for network-awareness and its implications on system and/or operating system design.

To make efficient use of the scarce and often widely fluctuating network resources available to mobile clients, Noble et al. [12] advocate a collaboration between operating system and application (on such mobile devices) to support concurrency of network-aware applications. Thereby the applications have a say on the (adaptation) policy, but as they cannot accurately measure their environment (because of concurrency issues), they must rely on system support and centralized resource management to resolve conflicts between applications.

Fox et al. [7] propose the use of scalable network services to solve the problem of scalability that infrastructural transcoding proxies witness when large numbers of users request network-aware content delivery services. Our study did not investigate the influence of high loads (caused, e.g., by numerous requests) on the result; the basic assumption of network-awareness is that the network is the bottleneck. For scenarios with different limitations, other techniques must be explored.

Lowekamp et al. [9] argues that network-level resource information provides accurate and economical information about the properties of network resources. There is compelling evidence to exploit network-level information in the context of a system like Remos that can obtain additional benefits from network level information. However, without the infrastructure, it is not clear how network-level information can be obtained; the paper reports data only for LAN experiments. In the Remos system, information for WAN connections is obtained through benchmarks [11].

6 Conclusions

To allow the network-aware application to make low-overhead adaptation decisions, the application requires access to information that is readily available in today's protocol stacks but is currently not exported (or exposed) to the application.

Network-awareness is (and will continue to be) important to mitigate the problems incurred by the heterogeneity and the significant fluctuations of the available bandwidth present in the Internet. Timely and accurate information about network status is instrumental in making well-founded adaptation decisions. Transport-level monitoring can effectively and very efficiently satisfy the need for timely and accurate bandwidth information. We find that transport-level monitoring is simple to implement and compares very favorably to application-level monitoring. These observations have direct implications for protocol/OS design: it is desirable to widen the protocol API to allow applications efficient access to the network status information required for adaptation.

Acknowledgements

We thank Peter Steenkiste for comments about the presentation of the material.

References

- [1] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Request for Comments (RFC 2475), Internet Engineering Task Force, December 1998.
- [2] J. Bolliger. *A framework for network-aware applications*. PhD thesis, ETH Zürich, April 2000. No. 13636.
- [3] J. Bolliger, T. Gross, and U. Hengartner. Bandwidth modelling for network-aware applications. In *Proceedings of IEEE Infocom '99*, pages 1300–1309, March 1999.
- [4] J. Bolliger, U. Hengartner, and T. Gross. The effectiveness of end-to-end congestion control mechanisms. Technical Report 313, Dept. Computer Science, ETH Zürich, February 1999.
- [5] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proceedings of IEEE Infocom 2000*, pages 1742–1751, March 2000.
- [6] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the remos system. In *Proc. 10th IEEE Symp.*

High-Performance Distr. Comp., San Francisco, CA, August 2001. IEEE CS Press.

- [7] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium of Operating System Principles*, pages 78–91, October 1997.
- [8] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. *Cluster Computing*, 2(2):139–151, 1999.
- [9] B. Lowekamp, D. O’Hallaron, and Th. Gross. Direct queries for discovering network resource properties in a distributed environment. *Cluster Computing*, to appear, 2000. An earlier version appeared in Proc. 8th IEEE Symp. High-Performance Distr. Comp.
- [10] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communication Review*, 27(3):67–82, July 1997.
- [11] N. Miller and P. Steenkiste. Collecting network status information for network-aware applications. In *Proceedings of IEEE Infocom 2000*, pages 641–650, March 2000.
- [12] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium of Operating System Principles*, pages 276–287, October 1997.
- [13] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM ’98*, pages 303–314, August 1998.
- [14] R. Weber, J. Bolliger, T. Gross, and H. Schek. Architecture of a networked image search and retrieval system. In *Proceedings 8th Intl. Conference on Information and Knowledge Management (CIKM ’99)*, pages 430–441. ACM SIGMIS/SIGIR, November 1999.
- [15] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *J. Future Generation Computing Systems*, 15(5-6):757–768, October 1998. Published also as UCSD Technical Report Number CS98-599.
- [16] G. R. Wright and R. W. Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, January 1995.
- [17] L. Zhang, S. Berson, S. Herzog, and S. Jamin. ReSource reservation Protocol (RSVP)— version 1 functional specification. Request for Comments (RFC 2205), Internet Engineering Task Force, September 1997.