The Implication of Network Performance on Service Quality: Testbed and Applications

Yu Zhang, Jia Wang, Srinivasan Keshav

Abstract—**As the Internet infrastructure evolves to include Quality of Service (QoS), it is necessary to map application quality requirements to network performance specifications in terms of delay and loss rate. In this paper, we first propose a new testbed by using a combination of simulation and emulation technics. Next, we use this testbed in studying the effect of network performance metrics on the perceptual quality of various applications to demonstrate the applicability of the testbed. We examine two typical applications on the basis the new testbed: the World Wide Web and the Internet Telephony. In our study of the World Wide Web, we derived a new TCP short connection model that computes the latency of Web retrieval accurately and efficiently given only the packet delay and loss rate characteristics *apriori*. In the study of Internet telephony, we show that the packet delay variance is the dominant network characteristics which affect the perceptual quality. As a result, the service quality drops dramatically when the network offered load reaches 80%. This can serve as a guideline for studies towards improving service quality of Internet telephony.**

I. Introduction

The Internet is emerging as the single networking infrastructure that carries the data, audio, and video traffic. It has long been recognized that this convergence requires the Internet to provide QoS guarantees to users and applications. With the emergence of many Internet QoS models [2] [3] [4], an interesting research issue arises, that is, how to relate network performance parameters to the service quality observed by an application. Internet service classes are typically defined in terms of bounds on performance parameters such as bandwidth, delay, delay jitter and loss rate. However, the notion of quality in these models, in terms of network performance parameters, such as delay and loss rate, is not the ultimate service quality delivered to applications, such as the retrieval latency of a Web page experienced by a user. We need to bridge the gap between network-oriented performance measures and application-oriented performance parameters.

We began our study with designing a new experimental testbed. The testbed is a combination of emulation and simulation. The *Entrapid* [10] [13] is used to simulate the network environment with different performance parameters. The real applications (e.g., Web clients and servers) are connected to the simulated network using the *RealNet* technology provided in the *Entrapid*. The composite setting provides us a simple and accurate experimental environment to study the behavior of the effect of network performance parameters on the applications.

To demonstrate the applicability of the testbed, we examine two typical applications: the World Wide Web and the Internet telephony. The World Wide Web is of great interest to us since the Web traffic dominates on the Internet today, comprising 75% of the overall bytes [29]. We proposed a new TCP short connection performance model which estimates the retrieval latency of a HTTP request given only the requested file size and the network delay and loss characteristics *apriori*. We choose Internet telephony because it is expected to be an important application on next generation QoS-aware Internet. Due to its advantages over traditional telephone network in terms of lower price and richer functionality, it is even growing rapidly despite the lack of QoS support on the Internet [19]. We analyze the relationship between network performance and various factors which directly determine the perceptual quality of the Internet Telephony.

In our study, we consider delay and loss rate as the network performance parameters of interest. These are, for the most part, the performance parameters provided by most service classes. We assume that losses are random. Though packet losses at a queue are known to be correlated, packet losses from a single stream have been found to have little or no correlation [6]. Moreover, with more and more routers adopting RED instead of drop-tail dropping policy, bursty loss due to instantaneous queue occupancy is becoming less frequent and even packet losses at a queue are expected to exhibit less correlation.

The rest of the paper is organized as follows. Section II describes our testbed setting. Section III studies the performance of the Web application under different network performance parameters and proposes a new model to estimate the latency of Web retrieval. Section IV presents our experimental results and performance analysis of the Internet telephony. Section V compares our work with related work in the literature. We conclude our work in Section VI.

II. Experimental testbed setting

Our experimental environment is modeled by two components: the *application entities* (e.g., the clients and the servers in the World Wide Web application), and the *network* connecting them. From the point of view of the applications, the network between two application entities can be abstracted as a virtual link with its characteristics defined by the network performance parameters. Correspondingly, our testbed consists of a simulated network using the *Entrapid* simulator [10] [13] and real applications connected to the simulated network using the *RealNet* technology in the *Entrapid*. This composite setting has two advantages: (i) *Simplicity.* We eliminate the need to modify, recompile, and relink applications with the network simulator; (ii) *Accuracy.* Since application executables are not modified, the behavior of the application is exactly as it would be were it running on an actual network.

Moreover, using the *Entrapid* makes it easy to customize the simulated network by specifying the desired delay and loss rate characteristics. As an example, the Fast Ethernet Simulation [31] built in the *Entrapid* can, according to the configuration (e.g. the interface card buffer size) and the monitored LAN traffic condition (e.g. the offered load, the mean packet size), impose on the packets the same delay and loss rate as they would experience in reality.

Figure 1 shows the testbed setting. We denote each Ethernet interface by its IP address. The testbed consists of two real machines [1] with *Entrapid*[2] simulator running on one and the application entities running on the other. We use *Entrapid* to simulate the network environemnt we expect to examine. Inside the *Entrapid*, two virtual machines, m_0 and m_1, with a virtual link w in between are created to simulate the network. Using the *RealNet* technology [10] [13], we bind each virtual machine to a real network interface (i.e. m_0 to the interface 172.16.4.3 and m_1 to the interface 172.16.5.3). We also bind the application entites, which are running on the other machine, to specified network interfaces. For instance, we bind the application client to the interface 172.16.4.2 and bind the application server to the interface 172.16.5.2.

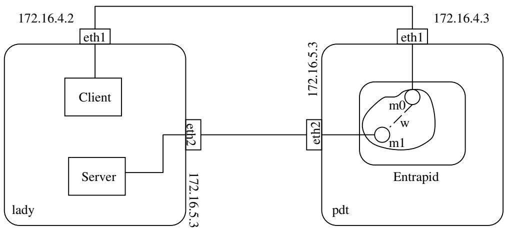

Fig. 1. The testbed setting: machine *lady* runs application entities and machine *pdt* runs the *Entrapid*. The routing tables are set up so that all the traffic between *lady* and *pdt* goes through the virtual link w in the *Entrapid*.

The routing tables at the network interfaces of these two machines are configured in such a way that all packets exchanged between the application entity 1 and entity 2 are enforced to go through the virtual link w. For example, in the Web application, a client request will traverse the interface 172.16.4.2, interface 172.16.4.3, virtual machine m_0, link w, vitual machine m_1, interface 172.16.5.3, interface 172.16.5.4 and arrives at the server. The server's reply will traverse all the way back to the client. By setting the delay and loss characteristics of the link w, we can simulate different network environment of interest and study the corresponding application behavior and quality.

The delay and loss characteristics to be imposed on the virtual link vary with different types of the network of interest (e.g. LAN, WAN, and wireless network). However, general conclusions about how delay and loss rate affect the application quality, drawn in one setting, can be used to infer the application performance in other settings once their delay and loss characteristics are known. In this work, we examine the application's performance mainly in a LAN environment. Yet the results collected and the conclusions drawn not only reveal the application performance in the Ethernet environment, but also shed light on how in general the application quality depends on delay and loss characteristics as well.

III. The World Wide Web Application

We study the performance of the World Wide Web under different network characteristicson the basis of the testbed proposed in Section II. The service quality of Web browsing is mainly measured by the retrieval latency of a HTTP request, defined as the client waiting period between the time the request is issued and the time the whole response is received. In our study, we use the transfer time of the underlying TCP connection as an approximation of the retrieval latency.

A. Methodology

The goal of our experiments is to study the behavior and performance of a Web traffic flow in the face of the changing delay and loss characteristics of the network environment. Our experiments focus on the effect of the network offered load on the Web retrieval latency because the offered load is considered to be the main contributor to the change of network performance characterized by delay and loss rate.

We use a simple HTTP browser as the Web client and Apache 1.3.4 as the Web server. Both the HTTP requests and responses will go through the virtual link in the *Entrapid*, which simulates a network[3] of specific delay and loss characteristics. The quality of Web service is then measured by the retrieval latency.

Each experiment includes multiple trials under a specific network configuration and Web page size. During each trial, the client retrieves a file (without embedded objects) of a fixed size from the server. The number of trials in each experiment is so determined to take both the time that a single trial takes and the stability of the results into account. Since the retrieval latency exhibits a large variance due to the various ways that the packet loss can affect the progress of a flow [9], a high stability (i.e. error $< 5\%$) of the results can only be achieved with at least 10,000 trials, which is infeasible in terms of the simulation time[4]. Fortunately, the number of trials on the magnitude of hundreds is good enough to capture the trend of the retrieval latency. The number of trials we choose for 3KB, 10KB, and 30KB files are 1000, 400, and 300, respectively.

In each trial, the Web client opens up a new TCP connection for the requested file. The transfer size on the connection is approximately the file size. We deliberately set the file size to capture the typical transfer size of the underlying TCP connection for Web retrieval, in order to illustrate the typical retrieval latency experienced by the user. We studied both HTTP/1.0 and HTTP/1.1. Our experiments on HTTP/1.0, the file sizes are chosen to be 3KB and 10KB[5], while in experiments on HTTP/1.1, it is chosen to be 30KB due to the use of persistent connection and request pipelining [6].

B. Experimental results

We vary the offered load[7] in the range of $0\% \sim 100\%$ and measure the resulting retrieval latency[8]. We observe that the effect of the offered load on the retrieval latency is a combination

[1] Both machines are Pentium II 333MHZ, with 128MB memory and 8Gbit hard disk, and running Linux 2.0.34.

[2] We use *Entrapid* 1.5 in our testbed setting.

[3] We simulate 10Mbps Ethernet using the Fast Ethernet Simulation [31] in the *Entrapid*.

[4] A typical trial for 30KB file with the network offered load equal to 90% may take around 200 seconds to complete.

[5] In generic HTTP/1.0 [5], the client opens up a new TCP connection to retrieve each object embedded in a Web page, which results in the median and average transfer sizes to be $2 \sim 3$KB and $8 \sim 12$KB, respectively [9].

[6] In HTTP/1.1, a persistent TCP connection can be used to retrieve several objects [12]. Pipelining allows the client to pipeline all the requests for the embedded objects in a page and the server to pipeline all the corresponding responses. The resulting average transfer size of a TCP connection, which can be roughly estimated as the average size of all the data on a web page, becomes $26 \sim 32$KB [18].

[7] The offered load is normalized by the network bandwidth.

[8] An overloaded network (i.e. the normalized offered load $> 100\%$) will cause a loss rate larger than 30%, which we believe is not a meaningful case to study.

of that of the packet delay and the loss rate. Furthermore, we decouple the effects of the packet delay and the loss rate by examining the individual effects and correlation between them.

As shown in Figure 2(a), the retrieval latency heavily depends on the network conditions. As the offered load increases, both the packet delay and the loss rate increase, which results in a rapid growth of the retrieval latency. Figures 2(b) and (c) show the individual effects of the packet delay and the loss rate on the retrieval latency, respectively. The latency increases approximately linearly with the increasing of the mean packet delay, but much more drastically with that of the loss rate. As the loss rate increases, the file transfer subjects to more severe TCP retransmission and congestion control to avoid instability. The increasing number of retransmissions and retransmission timer backoffs lead to a superlinear growth of the retrieval latency, which is the total time needed to deliver all the request and response packets.

Interesting enough, the effect of the offered load on the retrieval latency is approximately a straightforward summation of the individual effects of the packet delay and the loss rate. In other words, for a certain network offered load l, the resulting retrieval latency is approximately the sum of the latency caused by the packet delay corresponding to the offered load l and the latency caused by the loss rate corresponding to the offered load l. It implies that the effects of the packet delay and the loss rate are approximately independent of each other. In general, the effects of the packet delay and the loss rate are correlated. As we will see later, since the packet delay determines packet round-trip time (RTT), it plays an important role in TCP retransmission and congestion control. The independence we observed here is due to the LAN environment setting, in which the packet delay is relatively small even on a heavily loaded network and has a negligible effect on TCP retransmission and congestion control. However, the correlation exists in a WAN environment, where the packet delay can be as high as several seconds.

C. Modeling the performance of TCP short connections

Wealth of evidence suggests that TCP connections for HTTP requests are short, often around 10KB and often suffer high packet loss rates in the neighborhood of 5% [9]. We focus on deriving a TCP short connection model[9] to provide a quantitative explanation of the experimental results.

C.1 TCP

TCP (Transmission Control Protocol) [23] is a connection-oriented transmission protocol, which has been widely used to provide a reliable, ordered, bi-directional byte stream between two applications on Internet hosts. Mechanisms provided by TCP [1] [14] [16] [23] include window-based flow control and congestion control, sequence-number-based error control, acknowledgment, retransmission, and reassembly. While we provide a brief summary[10] here, a detailed description is available in [32].

Before sending any data, two endpoints must establish a connection by *three-way handshaking*. Once the connection has been established, the sender begins to send data packets[11], subject to the window-based flow control scheme. The sender keeps sending packets and putting them into the transmit queue[12] until outstanding data reaches the minimum of the *congestion window* ($cwnd$) and the *receiver-advertised window* ($rwnd$). After that, packets to be sent must be cached in the waiting queue. Upon receiving a data packet, the receiver acknowledges the highest in-order sequence number seen so far by sending an ACK to the sender. *Delayed ACK* algorithm is used to encourage ACKs to be piggybacked with data. To avoid excessive delay of ACKs, an ACK should be generated for at least every two full-sized segments, and must be generated within 500ms of the arrival of the first unacknowledged segment. Moreover, out-of-order data segments should be acknowledged immediately to accelerate loss recovery.

TCP detects network congestion by detecting packet loss and uses *retransmission* to handle packet loss. To adapt to the changing network conditions, the *retransmission timeout* (RTO) value is dynamically determined based on the estimation of the average and variance of RTT. A TCP sender employs a congestion control scheme to avoid causing network congestion by sending too fast [14]. Usually the sender is in one of two modes: slow start or congestion avoidance. In slow start mode[13], $cwnd$ starts at some minimum window and increases by one MSS for each ACK received that acknowledges new data. The sender enters slow start mode in three occasions: start of a connection, after a long idle time, and after a retransmission timeout, which implies a severe congestion [14]. Once $cwnd$ reaches the slow start threshold (*ssthresh*), the sender switches to more conservative congestion avoidance mode, where $cwnd$ increases by one MSS after receiving ACKs for all the packets in current $cwnd$. Whenever a retransmission timeout occurs, the sender takes it as a signal of network congestion. It immediately sets $cwnd$ to the initial minimum size, *ssthresh* to half of the outstanding data size, and enters into slow start mode.

C.2 Existing model for TCP short connection performance

The best-known model for a short TCP flow experiencing losses is proposed by [9] (referred to as *model C* in the rest of the paper). Since our model is evolved from this model, we briefly describe it here for completeness. We will also point out its limitations to motivate our improvements to the model.

C.2.a Model C. In model C[14], the data transfer is modeled as an initial connection establishment handshaking, followed by alternating phases of slow start and RTO runs, where an RTO run is a series of successive retransmission timeouts for one packet. This *alternating phase model* is illustrated in Figure 3 ([9]). The transfer time is $t = t_{handshk} + t_{RTO} + t_{xfer}$, where $t_{handshk}$, t_{RTO}, t_{xfer} are the time spent on handshaking, timeouts, and slow start respectively, and $t_{handshk} \approx RTT$.

Assume a TCP receiver that implements delayed acknowledgment sends one ACK for roughly every b ($b = 2$) packets. Let p be the packet loss rate, $data$ be the number of data segments to transfer, l be the number of losses during transferring this amount

[9] Our development and evaluation of the TCP short connection model are primarily based on the TCP specification of Linux 2.0.34.

[10] Parts of this summary is taken from [9].

[11] TCP gives each byte in the stream a unique 32-bit *sequence number*, and divides the sequence of bytes into segments of *maximum segment size* (MSS, typically 536 or 1460 bytes). Each segment, with a TCP header attached, is placed in an IP packet and sent to the receiver.

[12] The sender maintains two queues: the *transmit queue* for packets already sent but not acknowledged, and the *waiting queue* for packets to be sent.

[13] In particular, with delayed acknowledgment in effect, $cwnd$ grows exponentially by a factor of 1.5 [9].

[14] We provide a brief description of model C. For details, see [9] and [32].

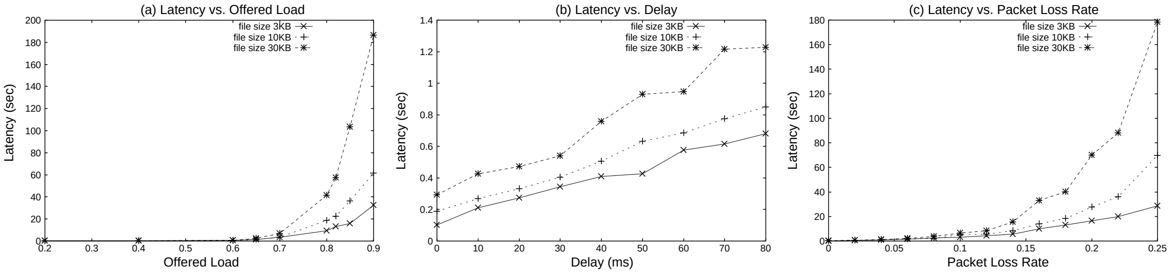

Fig. 2. The effect of the network performance on the retrieval latency for different file sizes: (a) retrieval latency vs. offered load, (b) retrieval latency vs. packet delay, (c) retrieval latency vs. packet loss rate.

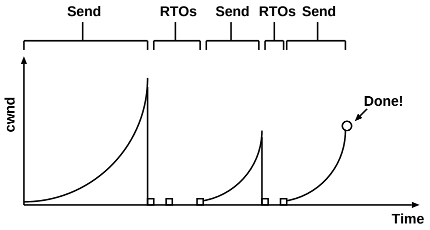

Fig. 3. Alternating phases of slow start and RTO runs.

of data, w be the initial $cwnd$ for the data transfer phase, RTT be the round trip time, and T_0 be the average start RTO value of an RTO run. The effective loss rate is $p = \frac{l}{data+l}$. So $l = \frac{p*data}{1-p}$. Model C uses the techniques proposed in [24] to derive the probability that a particular loss will incur a retransmission timeout[15]:

$$Q(p) = \begin{cases} 0 & \text{if } p = 0 \\ min\left(1, \frac{3}{\sqrt{\frac{8}{3bp}}}\right) & \text{if } p > 0 \end{cases}$$

Therefore, the number of retransmission timeouts experienced by a flow is $n = l \cdot Q(p)$. We take the loss rate p as the probability that RTO occurs for a send, so the number of RTOs in an RTO run is a geometric distribution with a mean of $\frac{1}{1-p}$, and the number of RTO runs, denoted by u, is $u = \frac{n}{\frac{1}{1-p}} = l \cdot Q(p) \cdot (1-p)$. The expected duration for a single RTO run is computed based on the assumption that the retransmission timeout is clamped down at $64T_0$, where T_0 is the first RTO value of an RTO run. The mean duration of an RTO run is $t_u = T_0 \cdot \frac{1+p+2p^2+4p^3+8p^4+16p^5+32p^6}{1-p}$ [32], and the total time spent on timeouts is $t_{RTO} = u \cdot t_u$.

Since the slow start phase are alternating with RTO runs, the number of slow start phases is $v = u + 1$. The average data sent per phase is $e = \frac{data+l}{v}$. Consider the progress of TCP in slow start mode in terms of *rounds* whose durations are equal to one RTT. Let $cwnd_i$ be the $cwnd$ of the sender at the beginning of round i. $cwnd_i$ is a geometric series with ratio $r = 1 + \frac{1}{b}$. and t_{xfer} can be estimated as $t_{xfer} = v \cdot \log_r\left(\frac{e(r-1)}{w} + 1\right) \cdot RTT$.

C.2.b Limitations. Unfortunately, model C has three major problems, which affect either its accuracy or its serviceability.

1. *Effective packet loss rate*. A data packet appears to be lost in two cases. First, the data packet gets lost (or overly delayed). Second, the data packet is delivered successfully, but its ACK (sent as a separate packet or piggybacked) gets lost (or overly delayed). The sender cannot distinguish these two cases and both of them can potentially trigger a retransmission timeout. Suppose each data packet triggers an ACK packet, the effective packet loss rate (i.e. the timeout probability) is $p = lossrate + lossrate * (1 - lossrate)$. Due to delayed acknowledgment, a data packet does not necessarily trigger an ACK, hence the computation of the exact effective packet loss rate is very complicated. However, we can use $p = lossrate + lossrate * (1 - lossrate)$ as a reasonable approximation.

2. *Estimating* t_{RTO}. First, Model C made some wrong assumption about the clamp-down retransmission timer. In FreeBSD, the retransmission timer is clamped down at $64IT_0$, where IT_0 is the initial RTT-based RTO value. Using Karn's algorithm [16], after an RTO run, the next packet is sent with the current backed-off retransmission timer in effect. We call it *cross doubling*, the doubling of the retransmission timer across consecutive data packets. So the first timeout T_0 of an RTO run is not necessarily IT_0, instead it can be multiple IT_0s. Using $64T_0$ as the clamp-down RTO value is an overestimation of the RTO values, which leads to overestimation of the transfer time especially when the packet loss rate is high and T_0 as multiple IT_0s is more likely to occur. Second, Model C computes the mean duration of an RTO run using T_0, the average first RTO value of an RTO run, and requires the value of T_0 to be measured. Since the value of T_0 depends on network performance, TCP retransmission and congestion control behavior, and the transfer size of the flow, usually it is different from one flow to another. Measuring T_0 for each flow is cumbersome and defeats the purpose of the model.

3. *Estimating* $t_{handshk}$. Model C computes handshaking time as a constant assuming no packet loss ever occur during handshaking. However, since the initial timeout for SYN packets is intentionally set to be large (e.g. 6 seconds in FreeBSD and 3 seconds in Linux), and the number of packets sent during handshaking is a nontrivial fraction of the total data packets sent for short connections, the time spent on timeout in handshaking is nonnegligible even for slightly high loss rate (e.g. $\geq 5\%$). This has been observed in both our traces and the simulations results in [9]. In fact, handshaking can be modeled as a special case of data transfer phase. We will discuss how we model handshaking in Section III-C.3.

Figure 4 shows how well model C estimates the transfer time of a fixed-size file for varying network offered load. The curve *measure* is the measured latency curve as in Figure 2. The curves *model C low* and *model C high* are both latency esti-

[15] While the probability model in [24] was for congestion avoidance mode, model C uses it to approximate the probability for slow start.

mated using model C with same $RTT = 2 * delay$, number of data packets $data = \frac{\text{file size + HTTP header size}}{1460}$, initial $cwnd$ $w = 1$, measured T_0, but different effective packet loss rates, i.e. $p = lossrate$ and $p = lossrate + lossrate * (1 - lossrate)$, respectively. Our observations are: (i) the absolute error of the estimated latency when the offered load $< 60\%$ is small; (ii) Model C underestimates the lstency due to the underestimation of the effective loss rate. Moreover, we notice that the effect of the underestimated loss rate outweighs the effect of the overestimated clamp-down timer; (iii) Model C with corrected effective loss rate overestimates due to the wrong modeling of the clamp-down timer [16].

Figure 5 shows how well model C estimates the file transfer time when the loss rate = 0 and the delay varies. Since the loss rate = 0, *model C low* and *model C high* both compute the latency as $t = t_{handshk} + t_{xfer}$, where $t_{handshk} = RTT$ and $t_{xfer} = \log_r \left(\frac{data\cdot(r-1)}{w} + 1\right) \cdot RTT$. So these two curves overlap and increase linearly with the delay. There is small overestimation observed, which is due to two factors: (i) Assuming every two data segments trigger one ACK is a conservative estimation of how fast an ACK is generated. In reality, ACKs may be sent back and $cwnd$ may open more frequently, resulting in faster data transfer; (ii) The initial $cwnd$ for the data transfer phase is not necessarily one MSS in Linux. Unlike FreeBSD, which fixes $cwnd$ to be one MSS during handshaking, Linux employs slow start and congestion avoidance during the handshaking as well as the data transfer phase. It turns out that the initial $cwnd$ can be two MSS with a considerable large probability (see Section III-C.3). Therefore, the data transfer could be faster in reality.

Figure 6 shows how well model C estimates the file transfer time when the delay = 0 and the packet loss rate varies. The observations are similar to those obtained from Figure 4 except that the absolute error of the estimated latency is small when the loss rate $< 5\%$.

C.3 Improving model C

We propose a new TCP short connection performance model to address the problems of model C. In this model, data transfer is still modeled as alternating phases, and t_{RTO}, $t_{handshk}$, t_{xfer} are computed separately. However, the effective packet loss rate is used throughout the estimation. The computation of t_{RTO} is modified to estimate T_0 and the clamp-down retransmission timer correctly. The computation of $t_{handshk}$ is modified to take timeouts during handshaking into account. We use the effective packet loss rate $p = lossrate + lossrate * (1 - lossrate)$ in our new model instead of $p = lossrate$ in Model C.

C.3.a Estimating t_{RTO}. A major improvement we made to model C is the way we compute t_{RTO}, the time spent on timeout during the data transfer phase. For the sake of simplicity, we ignore fast retransmission and fast recovery, and simplify delayed acknowledgment as the rule of generating one ACK per two segments as in model C.

The evolution of RTO value in Linux 2.0.34 implementation

[16]Note that the maximum RTO value in Linux is 120 seconds instead of $64T_0$ in model C. But since the range of T_0 we observed is 20 ms $\sim$ 3 second, $64T_0$ is not far from 120 seconds. The overestimation is mainly due to the problematic modeling of the clamp-down timer and the underestimation is mainly due to the wrong effective loss rate.

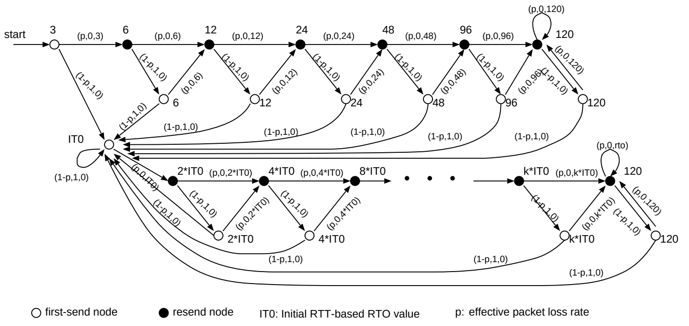

Fig. 7. The evolution of RTO value in the data transfer phase.

is depicted as a state trasaction diagram in Figure 7. Each node stands for a data sending (i.e. transmission or retransmission). The value associated with each node is the RTO value used to set the retransmission timer for that data sending. The start point corresponds to the first transmission of the first packet with the RTO timer set to 3 seconds. Associated with each edge is a tuple (*prob*, num_pkt, *time*), where *prob* is the probability of taking this edge, num_pkt is the number of packets delivered by taking this edge and *time* is the time added to the total timeout time by taking this edge. Each node with the RTO value rto can take one of two edges:

- *Timeout edge (p, 0, rto).* With probability p, the packet fails to be delivered and a timeout occurs. The rto will be added to the total timeout time. Upon a timeout, the RTO value doubles (i.e. retransmission timer exponential backoff). So the value associated with the node pointed to by this edge is $2 \cdot rto$.
- *Send-success edge* ($1 - p$, *1, 0).* With probability $1 - p$, the packet is delivered successfully, and the total timeout time remains the same. Two cases need to be distinguished. (i) The packet delivery succeeds on its first attempt. Upon receiving an ACK for this packet, the sender performs RTT estimation and updates the RTO value. Therefore, the RTO value for the end node of this edge should be an RTT-based RTO value IT_0. (ii) The packet delivery succeeds after more than one attempts. According to Karn's algorithm, the ACK for this packet does not trigger RTT estimation and RTO update. The RTO value for the end node of this edge is still rto. We determine which case a node is in when it takes a *send-success* edge by examining the edge through which it is reached. If the edge is a *send-success* edge, the node is a first transmission of a packet (*first-send* node depicted as a circle in Figure 7) and it is in Case (i). If the edge is a *timeout* edge, the node is a retransmission (*resend* node depicted as a dot in Figure 7) and it is in Case (ii).

Since the maximum RTO value is clamped down at 120 seconds, the "doubling chain" of RTO value is finite. A typical RTO run starts from a *first-send* node, followed by one or more *timeout* edges and *resend* nodes, and ends with a *send-success* edge and a *first-send* node. T_0 is the RTO value associated with the start node in an RTO run. For simplicity, we assume that $srtt = RTT = 2 * delay$ and $mdev = \frac{srtt}{2}$. Since the Linux implementation specifies the minimum RTO value to be 200ms, the RTT-based RTO value is a constant $max(srtt + 4 * mdev, 200ms) = max(3 * srtt, 200ms) = max(6 * delay, 200ms)$.

However, it is hard to get the analytical form of t_{RTO}. If we

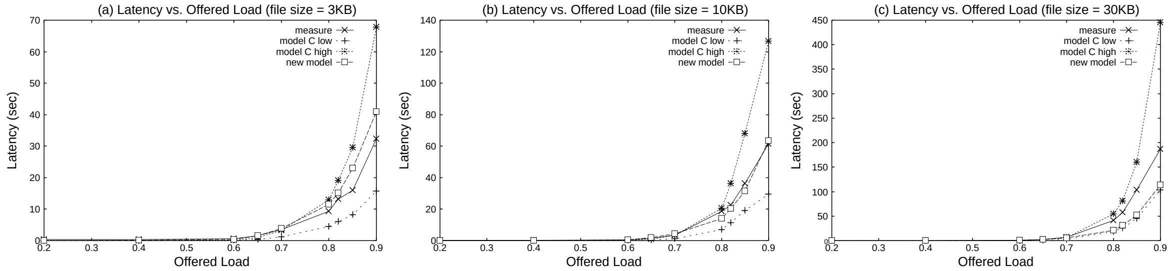

Fig. 4. Offered load vs. retrieval latency: (a) file size = 3KB, (b) file size = 10KB, (c) file size = 30KB.

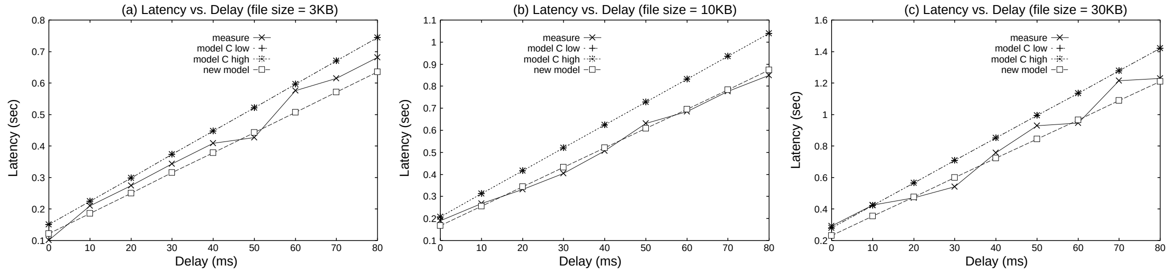

Fig. 5. Packet delay vs. retrieval latency (with the packet loss rate set to 0): (a) file size = 3KB, (b) file size = 10KB, (c) file size = 30KB.

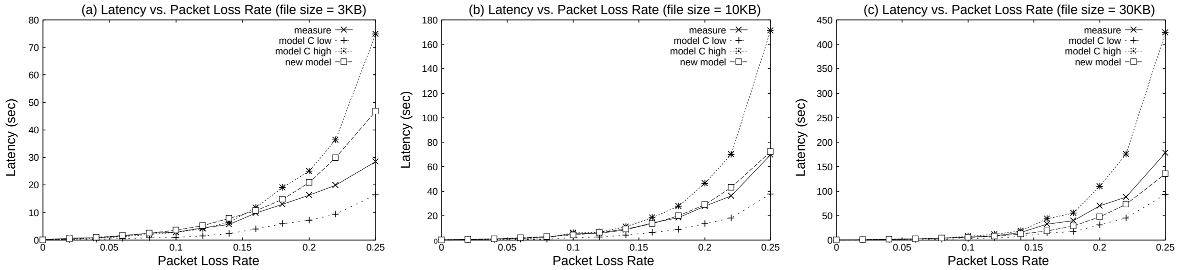

Fig. 6. Packet loss rate vs. retrieval latency (with the packet delay set to 0): (a) file size = 3KB, (b) file size = 10KB, (c) file size = 30KB.

can compute the expected duration of a single RTO run t_u, then by using the expected number of RTO runs u computed as in Section III-C.2, we can approximate t_{RTO} as $u \cdot t_u$. Let T_{0m} denote the first RTO value for an RTO run of data packet m, t_{um} denote the expected duration of this RTO run. Our further analysis on Figure 7 shows that $T_{0m} = f(T_{0(m-1)})$, $t_{um} = E[h(T_{0m})]$, and $t_u = \frac{1}{data}\sum_{m=1}^{data} t_{um}$, where f and h are non-linear functions. So $t_{RTO} = E[g(T_{01}, T_{02}, \cdots, T_{0,data})]$, where g is a complicated non-linear function, and $T_{01}, T_{02}, \cdots, T_{0,data}$ are random variables whose distributions are unknown.

In principle, once we have the evolution graph of the RTO value, given the number of data segments to send, denoted by $data$, we can compute t_{RTO} as follows. Let P denote the set of all paths with the sum of num_pkt of all edges along each path equal to $data$. Each path represents a possible way timeouts occur during the data transfer. So the expected timeout time for the data transfer is

$$\begin{aligned} t_{RTO} &= \sum_{\text{path p} \in \text{P}} \text{the probability of taking } p * \text{the timeout time spent on } p \\ &= \sum_{\text{path p} \in \text{P}} \left(\prod \textit{prob} \text{ of edges along } p\right) * \left(\sum \textit{time} \text{ of edges along } p\right) \end{aligned}$$

In effect, this is equivalent to the simulation approach, which is usually used when we are interested in computing $\theta = E[g(X_1, X_2, \cdots, X_n)]$, where g is some specified function, but it is not possible to analytically compute θ. Using this approach, we generate sufficiently many combinations of $X_1, X_2, \cdots, X_n$, compute the corresponding $g(X_1, X_2, \cdots, X_n)$, and use the average of these function values as an estimate of θ [26]. In our case, we don't have analytical form of the function g, but we can compute the function value using Figure 7. By simulating a whole run of Figure 7 sufficiently many time (where a run starts with sending the first packet and ends when number of $data$ packets are sent successfully), and measuring the RTO time in each run, we effectively generate sufficiently many combinations of T_{01}, T_{02}, $\cdots$, $T_{0,data}$, and compute $g(T_{01}, T_{02}, \cdots, T_{0,data})$. We can use the average RTO time per run as a reasonable estimate for t_{RTO}. By using this simple and efficient approach, we are able to avoid the complexity of real TCP simulator, and work around the intricacy of deriving a pure analytical model as well.

C.3.b Estimating $t_{handshk}$. In addition, we improved the estimation of the handshaking time by taking timeouts during handshaking into account. Figure 8 shows the 3-way handshaking on the TCP connection for a Web retrieval session. The case when no packet is lost during this phase is depicted in Figure 8(a). The Web client initiates the connection establishment by sending a SYN packet to the Web server. On receiving the SYN, the server sends its own SYN with an ACK for the client's SYN piggybacked (i.e. the SYN/ACK packet). When the SYN/ACK reaches the client, it triggers a separate ACK packet from the client to the server. At this point, the client enters into ESTABLISHED state. It sends out the first data packet (the HTTP request) to the server with the ACK for the server's SYN piggybacked. Only when the server receives the ACK for its SYN either piggybacked or sent as a separate packet, will it enter into ESTABLISHED state and start to send data packets. In Linux 2.0.34 TCP implementation, the first SYN and SYN/ACK packets are both sent with the retransmission timer set to 3 seconds, while the ACK packet is sent without any retransmission timer set and sent only once.

Retransmission (and possible backoff) happens when there is packet loss. Whenever the SYN packet or the SYN/ACK packet gets lost, it will be retransmitted and the retransmission timer keeps backing off until it is finally delivered (Figure 8(b) and (c)). If the last ACK packet gets lost, the client counts on the first data packet sent to the server to piggyback the ACK. As usual, the data packet may be retransmitted several times before it reaches the server (Figure 8(d)).

Thus, we estimate $t_{handshk}$ as $t_{handshk} = ths_{xfer} + ths_{RTO}$, where ths_{xfer} is the transfer time of the packets during handshaking, and $ths_{xfer} \approx 1.5 * RTT$ and ths_{RTO} is the time spent on timeout during this phase.

We can view the 3-way handshaking phase as the concatenation of two special data transfer phases whose t_{RTO}s can be efficiently estimated by simulation. The first phase is a reliable transfer of the client's SYN packet, including the (re)transmissions of SYN and SYN/ACK. Let ths_{RTO1} denote the timeout time in the first phase. It can be computed as t_{RTO} in a data transfer phase for one data packet with the effective loss rate $p = lossrate + lossrate * (1 - lossrate)$. The second phase can be one of two cases: (1) with probability $(1 - lossrate)$, it is a successful transmission of the ACK packet, with the timeout time equal to 0; (2) with probability $lossrate$, it is a loss of the ACK packet, followed by the (re)transmissions of the first data packet of the client until it successfully arrives at the server. Let ths_{RTO2} denote the timeout time in this case. It can be computed as t_{RTO} in a data transfer phase for one data packet with the effective loss rate $p = lossrate$ (since only the first data packet not its ACK need to be successfully delivered). Finally, the timeout time in the handshaking phase is $ths_{RTO} = ths_{RTO1} + lossrate * ths_{RTO2}$.

C.3.c Initial $cwnd$ of the data transfer phase. Finally, a minor modification to model C is regarding the initial $cwnd$ of the data transfer phase. Following FreeBSD, model C fixes the initial $cwnd$ to one MSS. However, in Linux 2.0.34, the initial $cwnd$ depends on the result of the slow start and congestion avoidance during handshaking. When the TCP connection is initialized, $cwnd$ is one MSS and $ssthresh$ is some very large number. With probability $(1 - p)^2$, no loss happens to SYN and SYN/ACK packets during handshaking, $cwnd$ becomes two MSS and $ssthresh$ remains the same for both the client and the server when they start sending data packets. Once either SYN or SYN/ACK ever gets lost, $cwnd$ becomes one MSS and $ssthresh$ becomes zero upon the retransmission timeout and $cwnd$ remains one MSS till the end of handshaking. Thus the initial $cwnd$ size w (measured by the unit of MSS) used in t_{xfer} computation should be $w = (1 - p)^2 \cdot 2 + (1 - (1 - p)^2) \cdot 1$.

C.3.d Summary of the new model. To sum up, our new model is given by the following analytical equations

$$
\begin{aligned}
p &= lossrate + lossrate * (1 - lossrate) \\
t &= t_{handshk} + t_{RTO} + t_{xfer} \\
l &= \frac{p * data}{1 - p} \\
Q(p) &= \begin{cases} 0 & \text{if } p = 0 \\ min\left(1, \frac{3}{\sqrt{\frac{8}{3bp}}}\right) & \text{if } p > 0 \end{cases} \\
u &= \frac{n}{\frac{1}{1-p}} = l \cdot Q(p) \cdot (1 - p) \\
v &= u + 1 \\
e &= \frac{data + l}{v} \\
r &= 1 + \frac{1}{b} \\
t_{xfer} &= v \cdot \log_r \left(\frac{e(r-1)}{w} + 1\right) \cdot RTT \\
t_{handshk} &= ths_{xfer} + ths_{RTO} \\
ths_{xfer} &\approx 1.5 * RTT \\
ths_{RTO} &= ths_{RTO1} + lossrate * ths_{RTO2} \\
w &= (1 - p)^2 \cdot 2 + (1 - (1 - p)^2) \cdot 1
\end{aligned}
$$

where t_{RTO} is computed as

$$
\begin{aligned}
t_{RTO} &= \sum_{\text{path p} \in \text{P}} \text{the probability of taking } p * \text{ the time out time spent on } p \\
&= \sum_{\text{path p} \in \text{P}} \left(\prod \textit{prob} \text{ of edges along } p\right) * (\ sumtime \text{ of edges along } p)
\end{aligned}
$$

by simulation with P denoting the set of all paths with the sum of num_pkt of all edges along each path equal to $data$, and the effective loss rate being $p = lossrate + lossrate * (1 - lossrate)$. ths_{RTO1}, ths_{RTO2} are computed similarly with the number of data packets being 1, and the effective loss rate being $lossrate$.

By closely simulating TCP retransmission behavior, including retransmission timeout probability, RTO backoff and clampdown, timeouts and $cwnd$ change during handshaking, our model is more accurate in estimating the transfer time. In addition, by eliminating the need to measure T_0, our model has better serviceability.

C.3.e Performance.

- *Accuracy*. Figures 4, 5, and 6 compare the accuracy of our new model with that of model C for transferring a fixed-size file with varying load, loss rate, and delay. From Figure 5, we can see that the new model estimates t_{xfer} in a way similar to model C and their estimations are both very close to the measurement results. The new model is more accurate than model C in estimating t_{RTO}, as shown in Figure 6. As a result, the new model is more accurate in estimating the transfer time with varying load. Compared with the curves of the measurement results, when loss rate is high (i.e. $> 10\%$), the new model overestimates for small transfer sizes (i.e. $\sim$ 3KB) and underestimates for large transfer sizes (i.e. $\sim$ 30KB), but fits well with moderate transfer sizes (i.e. $\sim$ 10KB) (we will discuss the possible reasons in Section 3.5.5). The curve of the new model in Figure 4 exhibits as the combination of corresponding curves in Figures 5 and 6, hence it also overestimates for small transfer size and underestimates for large transfer size when the load is high (i.e. $> 70\%$).
- *Complexity*. The most time-consuming part of the new model is the computation of t_{RTO} by simulation. The RTO simulator implements a random function g (Section III-C.3) and needs

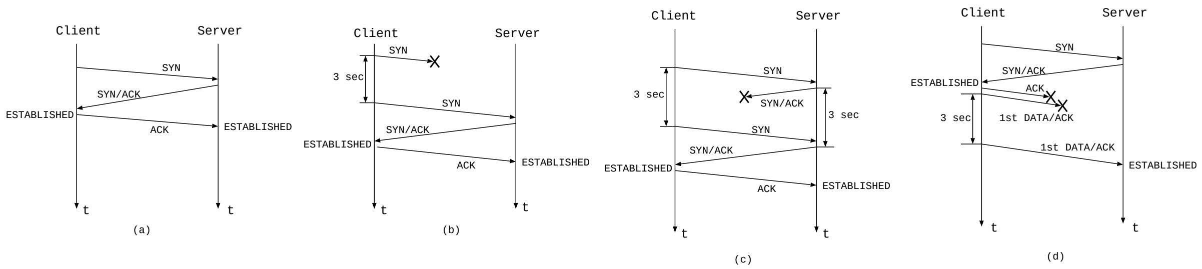

Fig. 8. Four cases of the 3-way handshaking: (a) no packet loss, (b) the SYN packet is lost, (c) the SYN/ACK packet is lost, (d) the last ACK is lost.

to be executed sufficiently many times to compute the expected value. Suppose the RTO simulator needs to be executed N times to get a stable expected value, and a pure analytical model such as model C costs time T, the new model costs about $N \cdot T$ in time. We observed that $N = 10000$ is sufficient to generate an expected value with error $< 5\%$ and $N = 100\times$ is good enough to capture the trend of t_{RTO}. Since T is very small, the speed of the new model is not a concern.

C.3.f Discussions. As model C, our new mode also makes some simplifications to render modeling TCP performance feasible. These simplifications introduce some inaccuracy to the model.

- *Effect of the transmit queue*. Assuming packet loss only triggers retransmission timeout, both model C and the new model take data transfer as a handshaking followed by alternating phases of slow start and RTO runs. Once a retransmission timeout for a packet occurs, an RTO run begins. However, as we observed from the simulation traces, the demarcation between slow starts and RTO runs is not so clear-cut. Often multiple packets are sent out and put into the transmit queue at the same time, and several packets can be lost concurrently. The transmit queue has a great impact on how timeouts of the packets in the queue contribute to the total timeout time.

<u>Observation</u> *Suppose the transmit queue has length* > 1. *If the first packet in the transmit queue times out, some of the subsequent packets, if lost, will get a resend for free, i.e. the resend is not triggered by a retransmission timeout, so neither will it trigger a* $cwnd$ *backoff, nor will any* RTO *value be added onto the total timeout time.*

When a transmit queue of data packets are sent out, the retransmission timer is only set for the first packet in the queue. This first packet keeps being (re)transmitted until an ACK for it arrives. If some subsequent packet in the queue gets lost, the ACK will not acknowledge all the packets in the queue. Upon receiving such an ACK, we infer that some of the unacknowledged packets get lost. To speed up loss recovery, we retransmit the packets in the transmit queue until outstanding data reaches min($cwnd$, $rwnd$). These packets form the updated transmit queue, and the retransmission timer will be adjusted for the first packet in this queue again.

Our RTO simulator omits the effect of the transmit queue, examining packets one by one and adding their timeout time up. This is merely a rough approximation which causes overestimation on t_{RTO}. Moreover, the effect of the transmit queue becomes more conspicuous when the packet loss rate is high.

Complexity arises if we want to modify our model to incorporate the effect of the transmit queue. In particular, we need to: (1) keep track of the state of the transmit queue, including simulating its initialization and shift when old packets get acknowledged and new packets are sent out; (2) keep track of $cwnd$ and the related information, i.e. simulate slow start and congestion avoidance, in order to get the limit of outstanding data size when we manipulate the transmit queue; (3) simulate cumulative and delayed ACK. We need to know when an ACK (sent as a separate packet or piggybacked) arrives in order to trigger the update of $cwnd$ related information and the shift of the transmit queue.

- *Delayed ACK*. Due to delayed ACK, $srtt$ in reality is an estimation of RTT plus the delay added for each acknowledgment. Though TCP avoids excessive delay of ACK to prevent large skew in RTT estimation, delayed ACK has a nonnegligible effect on RTT estimation, hence the computation of RTO value. For instance, when transferring a file of size 10KB with 20% packet loss rate and zero packet delay, we expect every RTT-based RTO value is the minimum RTO value (200ms), if the delay for the ACKs are negligible. But the simulation trace shows that 32% of the RTT-based RTO values are actually in the range of 200ms $\sim$ 3 seconds. Moreover, the effect of delayed ACK increases when the transfer size becomes larger. With other conditions being the same, if the transfer size becomes 50KB, 68% of the RTT-based RTO values are more than 200ms. So omitting the delay for ACKs leads to underestimation on the RTO values, hence the underestimation on the transfer time, and the underestimation becomes more severe when the transfer size increases.

- *Fast retransmission and fast recovery*. Like model C, our model assumes packet losses only trigger retransmission timeout. Model C states that "short flows typically never have $cwnd$ big enough for fast retransmit and fast recovery to happen" [9]. By examining the *tcpdump* trace in our experiments, we found that this claim only holds when the transfer size is very small (e.g. 2$\sim$3KB), or the packet loss rate is large enough (e.g. $\geq 15\%$). If the loss rate is small, a TCP flow transferring 10KB file witnesses considerable occurrences of fast retransmit and fast recovery. Fortunately, with a low loss rate, the transfer time for a short flow is small, and the absolute error caused by ignoring fast retransmit and fast recovery is small. Thus it is a reasonable simplification to ignore fast retransmit and fast recovery in the performance model, which results in a slight overestimation on the transfer time when the loss rate is small.

The above three simplifications introduce some inaccuracy into our model. We conjecture that for small transfer size, the effect of transmit queue dominates among these factors, so our model appears to overestimate the transfer time. For large transfer size, the effect of delayed ACK dominates, so the total effect becomes underestimation. For moderate transfer size, the effect of transmit queue offsets that of delayed ACK, so our estimation is very close to the measurement results. Further study needs to be done to analyze and quantify the inaccuracy of our model, and to improve it based on the results.

IV. The Internet Telephony Application

Service quality of Internet telephony is the perceptual quality, which is affected by the proportion of audio packets successfully received and played, the playout delay, and human sensitivity to various acoustic effects. In this section, we study how packet delay and loss rate, as representative network performance characteristics, affect the perceptual quality of Internet telephony application.

A. Experiments

As before, our experiments began with the testbed setting. The application entities in our testbed are two data exchange components which act as both senders and receivers. As a sender, it samples voice data periodically from an input source (e.g. file or microphone), packetizes it, sends it over the network. As a receiver, it reconstructs the incoming voice packets, plays it out in order with tolerable latency.

The data exchange components use an *adaptive playout* algorithm (algorithm 2 in [20]) to adjust the playout delay per talkspurt based on the estimation of packet delay and its variance. To adapt to a network spike[17] quickly, the algorithm has two modes of operation. In the normal mode, the estimate of packet delay is computed using running average. During a spike, it follows the rapid change of the packet delay[18].

The data exchange components also use a simple form of FEC [8] to handle packet loss. Each packet sent includes the current sample (in primary encoding) and redundant information of the previous sample (in secondary encoding), which may be stored upon receiving and played out when the primary sample is not available. The choice of secondary encoding is a trade-off between the additional load that FEC put on the network and the quality improvement. In our implementation, we use PCM and ADPCM as primary and secondary encoding, respectively.

The delay and loss characteristics imposed on the network are the same as in the study of Web application (Section III). Since we use UDP as our transport protocol, packets may be lost on the transport layer due to various reasons. One typical reason that the received datagrams will be dropped is that the input socket buffer is full [28].

The perceptual quality of the playout of a stream of voice packets sent is a combination of various factors:

- *How many samples are played.* A sample may be lost or be delayed and late for its scheduled playout. Note that with FEC used, its secondary sample may arrive in time and is played instead.
- *Quality of the played samples.* The quality of a sample is mainly determined by the encoding scheme used for the sample. For instance, a PCM sample has a better quality than an ADPCM sample.
- *Pattern that low-quality, or lost samples are distributed.* With the number of low-quality or lost samples being equal, a sparsely distributed pattern has a much better quality than a bursty pattern, due to the limited sensitivity of human ear to low frequency noise and very small period of silence [30].
- *Playout delay and its variation.* The *playout delay* per see, as long as it is less than the human-tolerable upper limit, has only a small effect on the perceptual quality. However, since human ear is very sensitive to the pauses between talkspurts, the variance of *playout delay* for talkspurts greatly affects the perceptual quality. Generally, the listener will feel disturbed when the pauses between talkspurts undergo unnatural large fluctuation frequently due to change of the *playout delay*.

We use the following metrics to measure the above factors:

- *Played ratio.* Defined as $\frac{\textit{number of samples played}}{\textit{number of packets sent}}$. With other factors being equal, the closer this ratio is to 1, the better the quality of playout will be.
- *Played ratio (primary)* and *played ratio (secondary).* Defined as $\frac{\textit{number of primary samples played}}{\textit{number of packets sent}}$ and $\frac{\textit{number of secondary samples played}}{\textit{number of packets sent}}$, respectively. They give an idea of the quality of the played samples. The sum of them is the *played ratio*.
- Mean and standard deviation of the *playout delay* for talkspurts.

The pattern that low-quality, or lost samples are distributed depends on the loss distribution and the delay distribution. The more sparse these distributions are, the lower impact low quality or lost samples will have on the perceptual quality. We ignore this factor here for simplicity. But how the delay and loss distribution affect this factor, and how it affects the perceptual quality need to be further studied. In addition, we use *saved ratio* to evaluate the effectiveness of FEC, defined as $\frac{\textit{number of secondary samples played}}{\textit{number of lost samples} + \textit{number of late primary samples}}$.

B. Results and discussions

As in the study of the Web application, we vary the normalized offered load in the range of $0\% \sim 100\%$ and measure the metrics of interest. To study the individual effect of loss rate, we fix the delay to 0, and measure the metrics under varying loss rate. To study the individual effect of delay distribution, we vary the load (thus changing the delay distribution) while fixing the loss rate to 0, and measure the resulting metrics[19].

Figures 9(a), (c), and (e) show how the offered load, the packet loss rate, and the delay distribution affect the played ratio and the saved ratio, respectively. Figures 9(b), (d), and (f) show how these three parameters affect the playout delay and its variance, respectively. When the offered load increases, the packet delay, its variance, and the loss rate increase. With the increase of delay variance, the playout delay also increases. Moreover, it changes more frequently and rapidly between different talkspurts, implying that its variation also increases. This can be seen from Figures 9(b) and (f), where the increase of delay variance with the increasing load is the major contributor of the increase of playout delay and its variation.

The effect of packet loss rate on the played ratio and the saved ratio is shown in Figure 9(c). Almost all the received packets are played due to small delay variance. The packets saved by FEC

[17] A network spike is a steep rise followed by a linear, monotonic decrease back to the normal level in the packet delay[20].

[18] Refer [32] for details.

[19] Note this is different from the experiment design for Web application. Since it is delay variance, not delay, that has a major impact on the quality of telephony application, using a flat delay instead of a delay distribution while fixing the loss rate is meaningless.

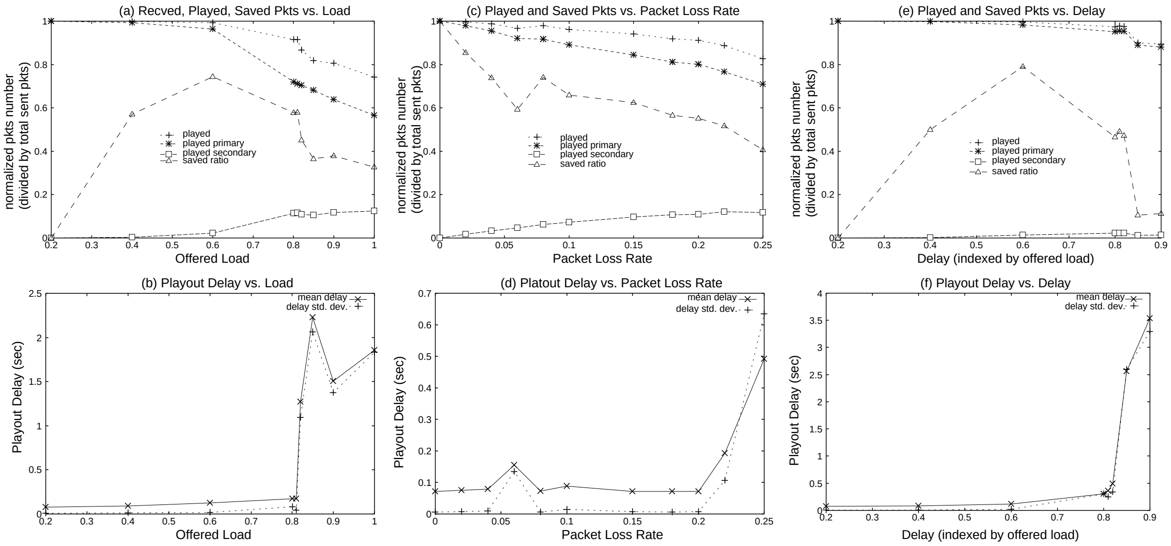

Fig. 9. The experimental results of Internet telephony: (a) played and saved ratios vs. offered load, (b) playout delay vs. offered load, (c) played and saved ratios vs. packet loss rate, (d) playout delay vs. packet loss rate, (e) played and saved ratios vs. packet delay (f) playout delay vs. packet delay.

are mainly lost packets. From the *saved ratio* curve, we can see that the ratio of lost packets saved by FEC is fairly high for all the loss rates between 0% and $25\%(> 40\%)$, implying that FEC is effective when packet losses are independent and non-bursty[20].

The perceptual quality degrades gradually when the packet loss rate increases linearly, with the ratio of played primary samples and the ratio of total played samples both decreasing. The quality stays bearable at surprisingly high loss rate until the loss rate reaches around 22%. This is mainly because human ears are more sensitive to the variation of playout delay than the randomly distributed lost and low-quality samples, and the variation of playout delay in this case is very small due to flat packet delay.

The effect of delay distribution on the played ratio and the saved ratio is shown in Figure 9(e). We fix the packet loss rate to be 0, so there is no loss introduced by the network. The evolution of the metrics exhibits two phases:

1. *Phase 1*: offered load $\leq\sim 80\%$. The delay variance is almost 0. With small delay variance, the received packets are also in order and evenly spaced, so UDP loss due to buffer overflow is rare. Due to small delay variance, few packets are late for playout. So *played ratio* and *played ratio(primary)* are almost 1. Moreover, the variance of playout delay is also very small (Figure 9(f))[21].
2. *Phase 2*: offered load $>\sim 80\%$. A steep rise of the packet delay variance not only causes more samples to be late for playout, but also causes more UDP losses due to buffer overflow. Thus the *played ratio(primary)* keeps going down after a sudden drop. As shown by the *saved ratio* curve, in this phase, FEC can only save a small portion of the late primary samples and lost samples, which we believe is due to the following reasons: (i) UDP loss is quite bursty, so the secondary sample for a lost primary sample is highly probable to be lost as well; (ii) With large delay variance, many of the secondary samples are late for playout themselves. Moreover, due to the steep rise of packet delay variance, the playout delay variance also increases suddenly and dramatically (the standard deviation $\approx$ the mean).

[20] We should ignore the start of the *saved ratio* curve, since during this period there are only few saved, lost or late primary samples, and the saved ratio is very sensitive to small changes of any of these numbers.

[21] Similarly, we should ignore the *saved ratio* during this phase.

Interestingly, though the packet loss rate is fixed to 0, the perceptual quality when we vary the delay distribution corresponding to increasing load degrades dramatically from phase 1 to phase 2, due to the large variation of playout delay and the sensitivity of human ears to pauses between talkspurts.

Figure 9(a) shows the effect of offered load on the played ratio and the saved ratio, which is a "deteriorated" combination of the individual effects of packet loss rate and delay distribution. By a "deteriorated" combination, we mean that the combined effect is more detrimental than the straightforward summation of the individual effects. Since large delay variance causes more secondary samples to be late for playout, fewer lost primary samples can be saved by FEC than in the fixed packet delay case (Figure 9(c)). The change of the played ratio and the saved ratio with increasing offered load exhibits three phases:

1. *Phase 1*: offered load $\leq\sim 60\%$. The packet delay variance and the loss rate are small. The network only introduces few losses. Small delay variance causes few UDP losses and few packets are late for playout. So *played ratio* and *played ratio(primary)* are almost 1. Meanwhile, the playout delay variance is small.
2. *Phase 2*: $\sim 60\% <$ offered load $\leq\sim 80\%$. The packet delay variance is still small so the late packets are still rare. But the packet loss rate starts increasing. The *played ratio* and *played ratio(primary)* start dropping. But since FEC is suitable to save randomly distributed packet losses, a considerable portion of lost samples ($>\sim 60\%$) are saved. Moreover, the playout delay variance is still small. So the perceptual quality degrades gradually during this phase.
3. *Phase 3*: offered load $>\sim 80\%$. Packet loss rate becomes high. A steep rise of the packet delay variance not only causes samples to be late for playout, but also causes many UDP losses. As shown by the *saved ratio* curve, the ability of FEC to save packets randomly dropped by the network are hampered by the large delay variance. So the *played ratio* and *played ratio(primary)* keep decaying after a sudden drop. Meanwhile, the playout delay variance increases suddenly and dramatically. As a

result, the perceptual quality degrades suddenly and dramatically from phase 2 to phase 3.

To sum up, our analysis of the relationship between network performance and the telephony perceptual quality shows that the variance of the playout delay dominates among all the factors which directly determine the perceptual quality. Accordingly, the packet delay variance dominates among network characteristics which affect the user-observed service quality of Internet telephony.

V. Related work

Previous work on study network performance and its effect on applications relies on purly simulation (e.g. NS simulator [22]) or real measurements. Our testbed is a combination of simulation and emulation, which is novel in this field to our best knowledge. No similar approach has been proposed in previous work.

On modeling TCP performance, most of the existing TCP performance models analyze the steady-state throughput of long bulk-transfer TCP connection with or without packet loss[11] [15] [17] [21]. The only model of TCP short connection performance is proposed in [9]. A lot of related work on Internet telephony focus on designing and improving specific techniques used in the telephony application, e.g. adaptive playout algorithm [20] [27], or FEC [7] [8] [25], which help to meet the service quality requirement in the face of changing network performance. From the clients' point of view, we put these techniques together in our telephony application, studying the relationship between the network performance and the perceptual quality when all the techniques come into play.

VI. Conclusions

In this paper, we designed a novel composite testbed which is a combination of simulation and emulation (i.e. it hooked real applications to the *Entrapid* [10] [13] network simulator). The former gives us manageability and efficiency by allowing us to impose changing network characteristics of various network environments, such as LAN, WAN, and wireless networks directly and easily, without sacrificing accuracy. The latter gives us great flexibility by allowing us to hook various kinds of applications to the simulated network as easily as in a real network environment.

To demonstrate the applicability of the testbed, we used it in studying the relationship between the network performance and the service quality of two representative Internet applications: the World Wide Web and the Internet telephony. Our study of Web applications shows that the retrieval latency increases linearly with the packet delay but nonlinearly with the packet loss rate. To provide a quantitative explanation, and to estimate the latency of Web retrieval given the requested file size, we improved the best known TCP short connection performance model [9] which estimates the transfer time of the requested file size on the underlying TCP connection. In our study of Internet telephony, we performed a qualitative analysis of the relationship between network performance and several factors which directly determine the perceptual quality. We showed that the variance of playout delay dominates among all the factors which directly determine the perceptual quality. This can serve as a guideline for studies towards improving service quality of Internet telephony.

References

[1] M. Allman, V. Paxson, and W. Stevens, TCP congestion control, Internet RFC 2581, Apr. 1999.

[2] Y. Bernet et al., A framework for differentiated services, Internet Draft *<draft-ietf-diffserv-framework-02.txt>*, Feb. 1999.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, An architecture for differentiated services, Internet RFC 2475, Dec. 1998.

[4] R. Braden, D. Clark and S. Shenker, Integrated services in the Internet architecture: an overview, Internet RFC 1633, June 1994.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk, Hypertext transfer protocol - HTTP/1.0, Internet RFC 1945, May 1996.

[6] J.-C. Bolot, Analysis of audio packet loss in the Internet, Proc. of Workshop on Network and Operating Systems Support for Digital Audio and Video, pp. 163-174, 1995.

[7] J.-C. Bolot, S. Fosse-Parisis, and D. Towsley, Adaptive FEC-based error control for Internet Telephony, Proc. of Infocom'99, Mar. 1999.

[8] J.-C. Bolot and A. Vega-Garcia, The case for FEC-based error control for packet audio in the Internet, ACM Multimedia Systems, 1997.

[9] N. Cardwell, S. Savage, and T. Anderson, Modeling the performance of short TCP connections, Technical Report, Computer Science Department, Washington University, Nov. 1998.

[10] *Entrapid* 1.5, *http://www.ensim.com*.

[11] S. Floyd, Connections with multiple congested gateways in packet-switched networks, part 1: one-way traffic, ACM Computer Communications Review, 21(5), Oct. 1991.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, Hypertext transfer protocol - HTTP/1.1, Internet RFC 2068, Jan. 1997.

[13] X. Huang, R. Sharma, and S. Keshav, The ENTRAPID protocol development environment, Proc. of Infocom'99, Mar. 1999.

[14] V. Jacobson, Congestion avoidance and control, Proc. of ACM Sigcomm'88, pp314-329, Aug. 1998.

[15] A. Kumar, Comparative performance analysis of versions of TCP in a local network with a lossy link, IEEE/ACM Trans. on Networking, 6(4), Aug. 1998.

[16] P. Karn and C. Partridge, Improving round-trip time estimates in reliable transport protocols, Proc. of ACM Sigcomm'87, pp. 2-7, August 1987.

[17] T. V. Lakshman and U. Madhow, The performance of TCP/IP for networks with high bandwidth-delay products and random loss, IEEE/ACM Trans. on Networking, June 1997.

[18] B. A. Mah, An empirical model of HTTP network traffic, Proc. of Infocom'97, Apr. 1997.

[19] N. F. Maxemchuk, Active networks in telephony, OpenArch'99, Mar. 1999.

[20] S. B. Moon, J. Kurose, and D. Towsley, Packet audio playout delay adjustment: performance bounds and algorithms, ACM/Springer Multimedia Systems, 5:17-28, Jan. 1998.

[21] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, The macroscopic behavior of the TCP congestion avoidance algorithm, ACM Computer Communications Review, 27(3):67-82, July 1997.

[22] UCB/LBNL/VINT Network Simulator - ns (version 2), *http://www.isi.edu/nsnam/ns/*.

[23] J. Postel, Transmission control protocol, Internet RFC 793, Sept. 1981.

[24] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, Modeling TCP throughput: a simple model and its empirical validation, Proc. of ACM Sigcomm'98, pp. 303-314, Aug. 1998.

[25] M. Podolsky, C. Romer, and S. McCanne, Simulation of FEC-based error Control for packet audio on the Internet, Proc. of Infocom'98, Mar. 1998.

[26] S. M. Ross, Introduction to Probability Models (6th Edition), pp598-599, Academic Press, 1997.

[27] R. Ramjee, J. Kurose, D. Towsley, and H. Schulzrinne, Adaptive playout mechanisms for packetized audio applications in wide-area networks, Proc. of Infocom'94, pp680-688, June 1994.

[28] W. Stevens, TCP/IP Illustrated, Vol. 2, Addison Wesley, 1995.

[29] K. Thompson, G. J. Miller, and R. Wilder, Wide-area Internet traffic patterns and characteristics, IEEE Network, 11(6):10-23, Nov. 1997.

[30] K. Toutireddy and J. Padhye, Design and simulation of a zero redundancy forward error correction technique for packtized audio transmission, Proj

[31] J. Wang and S. Keshav, Efficient and accurate Ethernet simulation, Proc. of the 24th Conference on Local Computer Networks (LCN'99), pp. 182-191, Oct. 1999.

[32] Y. Zhang, J. Wang, and S. Keshav, The implication of network performance on service quality, Technical Report TR99-1754, Department of Computer Science, Cornell University, July 1999.