# The Implication of Network Performance on Service Quality*

Yu Zhang, Jia Wang and Srinivasan Keshav
Cornell Network Research Group (C/NRG)
Department of Computer Science, Cornell University
Ithaca, NY 14853-7501
{*yuzhang, jiawang, skeshav*}@*cs.cornell.edu*

July 5, 1999

**Abstract**

As the Internet infrastructure evolves to include Quality of Service (QoS), it is necessary to map application quality requirements to the the network performance specifications in terms of delay and loss rate. While past work has addressed the dependency of audio and video applications on these underlying QoS metrics, little work has been done in the area of Web traffic. In this paper, we use a combination of emulation, simulation, and analysis to quantify the effect of network performance metrics on HTTP request latency and the perceptual quality of an audio application. Although our work is done in the context of a LAN environment, the results generalize to a more general WAN environment. Our contributions are three-fold. First, we combine simulation and emulation techniques in setting up an accurate yet controllable testbed. The use of network simulator *Entrapid* [10] [14] and its built-in Fast Ethernet Simulation [34] makes our simulation both efficient and accurate. Second, for Web applications, we define a new TCP short connection model that computes the latency of Web retrieval accurately and efficiently given only packet delay and loss rate characteristics *apriori*. Experiments show that our model significantly improves the accuracy of the best-known TCP short connection model by correctly capturing TCP retransmission behavior. Finally, for 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 Ethernet offered load reaches 80%. This can serve as a guideline for studies towards improving service quality of Internet telephony.

## 1 Introduction

The Internet is emerging as the single networking infrastructure that carries 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 service models (e.g. *Guaranteed Service* and *Controlled-load Service* provided in Intergrated Service model and *Premium Service* and *Assured Service* provided in Differentiated Service model) [2] [3] [5], 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. A

*This work was supported in part by NSF Grant ANI-9615811 and by a National Science Foundation Graduate Fellowship awarded to Jia Wang.

user/application belonging to a certain service class submits a traffic flow specification, and, in turn, receives the level of service defined by this service class. 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 address this issue by studying, in detail, the effect of network peformance parameters on two typical applications: the World Wide Web and the Internet telephony. The World Wide Web is of great interest to us since Web traffic dominates on the Internet today, comprising 75% of the overall bytes [32]. 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. 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 [20]. 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 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. For the purpose of this paper, we assume that if these parameters are not speficified directly, it is possible to transform the specified parameters to packet delay and loss feature. We further assume that losses are random. One justification for this is that with more and more routers adopting RED instead of drop-tail dropping policy, bursty loss due to instantaenous queue occupancy is becoming less frequent and packet loss is expected to exhibit less correlation.

We use both simulation and emulation in our study. We set up an experimental testbed where the *Entrapid* [10] [14] is used to simulate the network environment with different performance parameters. With this testbed, we examine the behavior of the World Wide Web and Internet telephony and how well they perform from the client's point of view. For the World Wide Web, we propose a new TCP short connection model to estimate the retrieval latency of a HTTP request given only the information of delay and loss characteristics *apriori*. For Internet telephony, we analyze the relationship between network performance and various factors which directly determine the perceptual quality.

The rest of the paper is organized as follows. Section 2 describes our experimental testbed setting. Section 3 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 4 presents our experimental results and performance analysis of the Internet telephony. Section 5 compares our work with related work in the literature. We end with concluding remarks and future work in Section 6.

# 2 Experimental testbed setting

In this section, we describe our experimental testbed. To study the behavior and performance of an application's traffic flow in the network environment, two components are needed in the experiments: the *application entities* (i.e. clients and servers in the World Wide Web and peers in Internet telephony), and the *network* connecting them (Figure 1). From the point of view of the applications, the network between two application entities is a virtual link with its characteristics defined by the network performance parameters.

Corresponding to the two conceptual components illustrated in Figure 1, our testbed has a network simulated

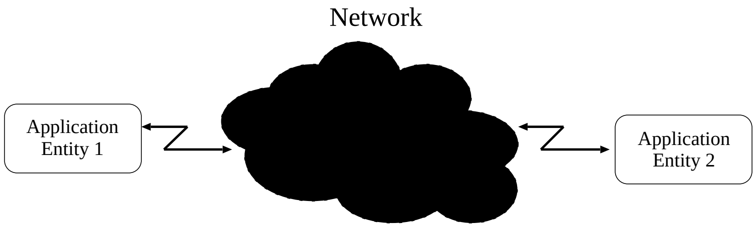


Figure 1: Two components in our experiments: the *application entities* present application behavior and the *network* is the connection between application entities.

using Entrapid simulator [10] [14] and real applications connected to the simulated network. This composite setting has two advantages.

1. *Simplicity.* We eliminate the need to modify, recompile, and relink applications with the network simulator;
2. *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.

We choose Entrapid [10] [14] for three reasons. First, it's suitable for studying behavior of protocols, such as TCP and UDP in a controlled fashion. In particular, it provides the abstraction of "a network in a box", in which users create, in a single user-level process, an entire network of virtual machines. Each virtual machine runs a separate OS kernel, which supports a full network stack from sockets to device driver.

Second, it supports RealNet technology which can seamlessly connect real-world devices such as routers and switches to the emulated network. So we can easily connect application entities to the "network box" using this technology in our composite experimental setting.

Finally, the modular design of the virtual link in 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 [34] built in Entrapid can according to the configuration (e.g. the interface card buffer size) and the monitored LAN traffic condition (e.g. the offered load, mean packet size), imposes on the packets the same delay and loss rate as they would experienced in reality. In addition, by presetting the traffic condition to reflect different LAN traffic instead of monitoring it, we are able to examine how the corresponding delay and loss characteristics affect the performance of a specific application without generating the real cross-traffic. This makes our simulation efficient and accurate.

Figure 2 shows the testbed setting. We denote the Ethernet interfaces by their IP addresses. Two machines have been set up with Entrapid 1.5 running on one and application entities running on the other. Both machines are Pentium II 333MHZ, with 128MB memory and 8Gbit hard disk running Linux 2.0.34. Inside Entrapid, two virtual machines, $m_0$ and $m_1$, with a virtual link $w$ in between are created to simulate the network. Using RealNet technology, we bind each virtual machine to a real network interface (i.e. $m_0$ to interface 172.16.4.1 and $m_1$ to interface 172.16.5.1). The routes are configured in such a way that all packets exchanged between the application entity 1 and entity 2 are forced to go through the virtual link $w$. By setting the delay and loss characteristics of $w$, 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 conclusion about how delay and loss rate affect application quality, drawn in one setting, can be used to infer the application performance in other settings

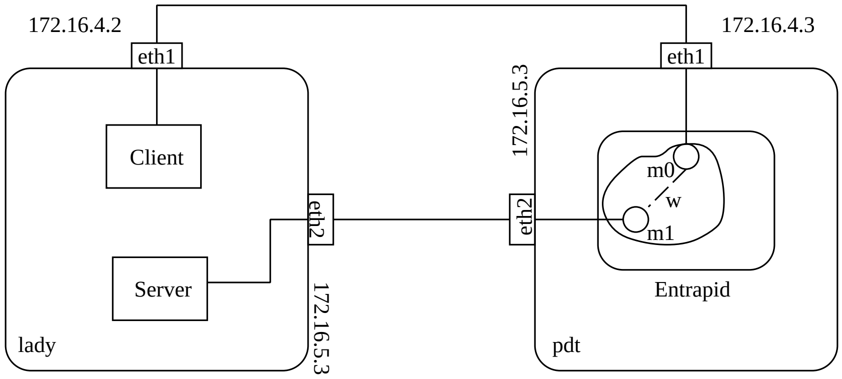


Figure 2: The testbed setting: machine *lady* runs application entities and machine *pdt* runs the *Entrapid*. Routes are set up so that all the traffic between *lady* and *pdt* goes through the virtual link $w$ in *Entrapid*.

once their delay and loss characteristics are known. Since delay and loss characteristics of Ethernet corresponding to different configuration and traffic conditions have been collected and incorporated into Entrapid [34], we take Ethernet as the network of interest in our study. Yet the results collected and the conclusions drawn not only reveal the application performance in Ethernet environment, but also shed light on how in general the application quality depends on the delay and loss characteristics in as well.

# 3 The World Wide Web

In this section, we study the performance of the World Wide Web under different network characteristics. We first present the simulation methodology and results. Since the retrieval latency of a Web request can be approximated by the transfer time of the underlying TCP connection, and this TCP connection is usually short, we focus on deriving a TCP short connection model to provide a quantitative explanation of the experimental results. We first introduce the best-known analytical model for TCP short connection performance, then we propose a new model to address its limitations. We will also compare the performance of these two models.

## 3.1 Methodology

The goal of our experiments is to study the behavior and performance of a Web traffic flow in the face of changing delay and loss characteristics of the network environment. Each experiment includes multiple trials under a specific network configuration and Web page size. During each trial, a simple HTTP/1.0 client retrieves a file (without embedded objects) of fixed size from the server. Both the HTTP requests and responses will go through the virtual link in *Entrapid*, which simulates a network of specific delay and loss characteristics. The quality of Web service is then measured by the retrieval latency. We use Apache 1.3.4 as the Web server.

### 3.1.1 Network performance characteristics

We simulate 10Mbps Ethernet using the Fast Ethernet Simulation [34] in *Entrapid*. Our experiments focus on the effect of network offered load on the Web retrieval latency, since the offered load is considered to be the main contributor to the change of network performance characterized by delay and loss rate. The relationship between the offered load and the packet delay and loss rate, incorporated in *Entrapid*, are shown in Figure 3.

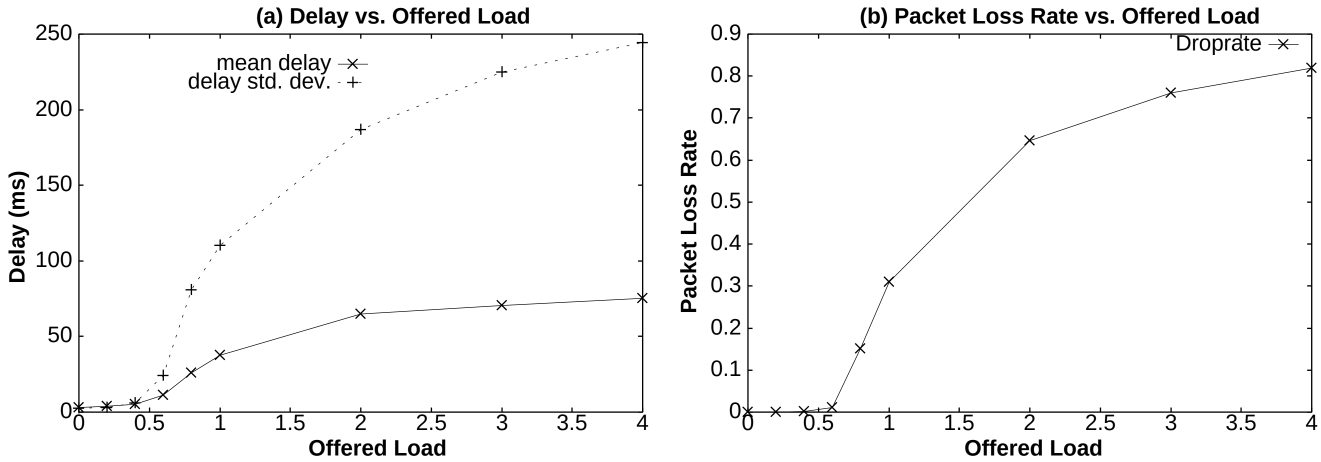


Figure 3: The network performance parameters configuration in our experiments (mean packet size = 512 bytes): (a) mean and variance of packet delay vs. offered load (b) packet loss rate vs. offered load.

### 3.1.2 Transfer size

In each trial, the HTTP/1.0 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. Both HTTP/1.0 and HTTP/1.1 are studied in our experiments. In generic HTTP/1.0, the client opens up a new TCP connection to retrieve each object embedded in a Web page [6], which results in the median and average transfer sizes to be 2-3KB and 8-12KB, respectively [9]. Correspondingly, our experiments on HTTP/1.0 are conducted on file sizes 3KB and 10KB.

The key differences between HTTP/1.0 and HTTP/1.1 are persistent connection and request pipelining. 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 26-32KB [19]. In our experiments on HTTP/1.1, the file size is chosen to be 30KB.

### 3.1.3 Number of trials

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 varied ways that 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 simulation time (a typical trial for 30KB file with network offered load 90% may take around 200 seconds to complete). 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.

### 3.2 Experimental results

We vary the offered load (normalized by the network bandwidth) in the range of $(0\%, 100\%)$ and measure the resulting retrieval latency[1]. We observe that the effect of the offered load on the retrieval latency is a combination of that of delay and loss rate. To further study the individual effects of the delay and loss rate respectively, and the correlation between them, we also measure the retrieval latency when we fix one of them and vary the other, thus decoupling their effects on the application performance.

As shown in Figure 4(a), the retrieval latency heavily depends on 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.

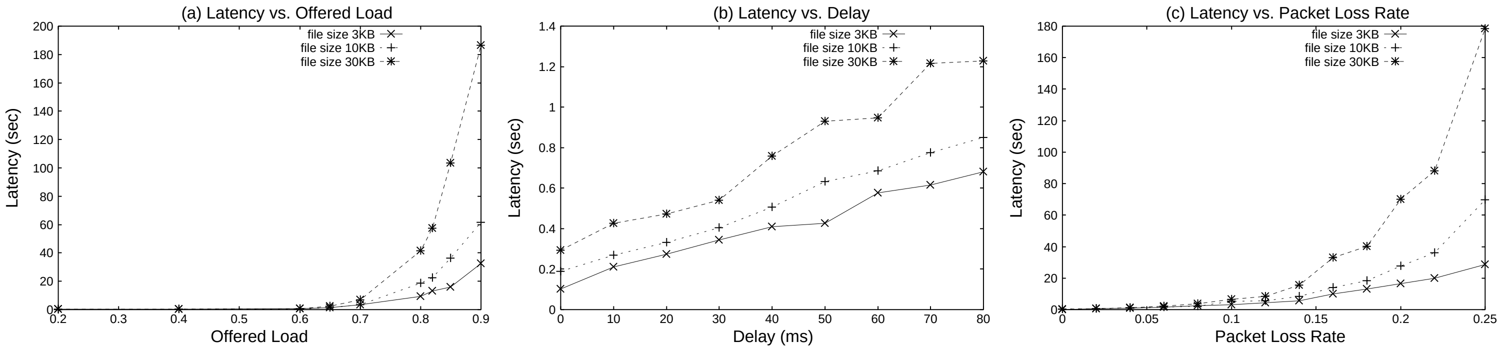


Figure 4: Effect of the network performance on the retrieval latency for different file sizes: (a) Retrieval latency vs. offered load, (b) Retrieval latency vs. packet delay, (c) Retrieval latency vs. packet loss rate.

Figures 4(b) and 4(c) show the individual effects of packet delay and loss rate on retrieval latency, respectively. The latency increases approximately linearly with the increasing of mean packet delay, but much more drastically with that of the loss rate. As the loss rate increases, the Web page transfer subjects to more severe TCP retransmission and congestion control to avoid instability. The increasing number of retransmissions and retransmission timer backoff lead to superlinear growth of the Web retrieval latency, which is in effect the total time needed to deliver all the request and response packets.

Interesting enough, the effect of network offered load on retrieval latency is approximately a straightforward summation of the individual effects of delay and loss rate[2], which suggests that the effects of delay and loss rate are approximately independent of each other. In general, the effects of delay and loss rate are correlated. As we will see later, since delay determines packet round trip time (RTT), it plays a role in TCP retransmission and congestion control. The independence we observed here is due to the LAN environment setting, in which delay is relatively small even on a heavily loaded network and has negligible effect on TCP retransmission and congestion control. However, the correlation exists in a WAN environment, where delay can be as high as several seconds.

### 3.3 TCP

Before providing further explanations of the experimental results, we first give an overview on TCP mechanisms[25] [1] [15] [17]. TCP (Transmission Control Protocol) [25] 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

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

[2] In other words, for a certain network offered load $l$, the resulting retrieval latency is approximately the sum of the latency caused by the delay corresponding to offered load $l$ and the latency caused by the loss rate corresponding to offered load $l$.

hosts. Mechanisms provided by TCP include window-based flow control and congestion control, error control based on sequence number, acknowledgement, retransmission, and reassembly.

### 3.3.1 General Mechanisms

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.

Before sending any data, a connection must be established between two endpoints by a *three-way handshaking*. First, the TCP connection initiator send a SYN packet with its *initial sequence number*(ISN) to the other side. Then the other side responds by sending a packet with an acknowledgment (ACK) for this ISN and an ISN of its own. Finally, the initiator sends an ACK packet acknowledging the other side's ISN.

Once the connection has been established, the sender begins to send data packets, which is controlled by a windowed-based scheme, i.e. the number of outstanding data (the data sent but not acknowledged) doesn't exceed the effective window size, which is the minimum of the *congestion window* (*cwnd*) and the receiver-advised window (*rwnd*). Upon receiving a data packet, the receiver acknowledges the highest in-order sequence number seen so far by sending an ACK to the sender. The TCP sender always sends as much data as its effective window allows and then waits for an acknowledgment. When it receives an acknowledgment, it sets *rwnd* to the window advertised in the ACK, increases *cwnd*, and again sends as much data as its effective window allows.

*Delayed ACK* algorithm is used to encourage ACK to be piggybacked with data. However, caution has been made to avoid excessive delay of ACK. 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, including those above a gap and those filling in all or part of a gap in the sequence space, should be acknowledged immediately to accelerate loss recovery.

TCP invokes congestion control schemes, which includes *slow start* and *congestion avoidance*, so that the sender avoids sending fast enough to cause network congestion and adapts to any congestion it detects by sending slower [15]. TCP detects network congestion by detecting packet loss and use *retransmission* to handle packet loss. Since our experiments on Web performance focus on examining behavior of short TCP connections. All TCP models presented in this paper involves study of congestion control and retransmission schemes, of which a detailed description is provided next.

### 3.3.2 Retransmission and Congestion Control

#### 3.3.2.1 Retransmission timeout

TCP uses retransmission to handle packet loss. One way to detect packet loss is through retransmission timeout (RTO). If an ACK for a packet is not received within a timeout interval (referred to as RTO value in this paper) , the packet is retransmitted. To adapt to the changing network conditions, RTO value is dynamically determined based on the estimation of both average and variance of RTT. RTT is sampled as the time between when a packet is sent and when its acknowledgement is received.

The sender maintains two queues: the transmit queue for packets already sent but not acknowledged, and the waiting queue for packets to be sent, submitted to TCP by user application. The sender will keep sending packets and put them into transmit queue until outstanding data reaches minimum of *cwnd* and *rwnd*. After that, packets to be sent must be cached in the waiting queue[3].

[3] For later discussion about the models, the retransmission and congestion control mechanisms we describe here is close to Linux

The retransmission timer is always so adjusted that the timeout will occur the time of latest RTO value after the first packet of the transmit queue is sent. The adjustment need to be performed whenever a new RTO value is computed, or the first packet of the transmit queue is changed. A timestamp recording sending time for each packet in the queue may facilitate the adjustment.

If consecutive retransmission timeouts occur for one packet, in the timeout handler, the RTO value will double, starting from the RTT-based initial value ($T_0$). To avoid instability of TCP performance due to extremely large RTO, maximum RTO value is clamped down either relative to initial RTO (e.g. $64T_0$ in FreeBSD) or by an absolute value (e.g. 120 seconds in Linux 2.0.34).

A lot of operations regarding retransmission occur upon receiving an ACK (either piggybacked or as a separate packet) for some data waiting to be acknowledged in the transmit queue:

- If we are not retransmitting, we will sample a new RTT value and perform RTT estimation, thus updating the RTO value. RTT is estimated as follows. Two state variables, *srtt* and *mdev* are maintained to estimate the mean value and variance of RTT respectively. And the RTO is computed as $RTO = srtt + 4 * mdev$ to adapt to the change of RTT and its variance. The first time we perform estimation, we sample a new RTT value in $m$, and let

$$srtt = m$$

$$mdev = \frac{m}{2}$$

  so that the first RTO is in fact $3m$. Subsequently, when we perform estimation, we compute the running averages of *srtt* and *mdev* values:

$$srtt = \frac{7}{8} srtt + \frac{1}{8} m$$

$$mdev = \frac{3}{4} mdev + \frac{1}{4}(m - srtt)$$

- If we are not retransmitting, since the ACK makes the sender window shift to right, we can take packets from the waiting queue, send them and put them into transmit queue, until outstanding data reaches min(*cwnd*, *rwnd*).

- If we are retransmitting, we don't perform RTT estimation to avoid skewing RTT estimation. In addition, the current backed-off RTO is kept for the next packet. Only when a subsequent packet is acknowledged without an intervening retransmission will the RTO be recalculated based on estimated RTT (Karn's algorithm [17]).

- If we are retransmitting, and there are packets in transmit queue waiting for acknowledgement, we can infer some of them (at least probably the first unacknowledged packet) get lost. To speed up loss recovery, we retransmit the packets in the transmit queue until outstanding data reaches min(*cwnd*, *rwnd*) (batch retransmission).

#### 3.3.2.2 Slow start and congestion avoidance

*Slow start* and *congestion avoidance* is the core mechanism of TCP congestion control [15]. It specifies how the sender congestion window (*cwnd*) evolves according to the observed network congestion state, thus controlling the flow from the sender to the receiver.

---

2.0.34 TCP implementation. But we believe the concepts should be echoed in typical TCP implementations.

TCP detects network congestion by detecting packet loss. Usually, packets can get lost due to damage in transit or buffer overflow along the transit path in a congested network. Since for most non-wireless networks, loss due to damage is rare($\ll 1\%$) [15], it is reasonable to take packet loss as a signal of congestion.

A TCP sender is in one of two modes: slow start or congestion avoidance. Slow start starts with a small sending rate and increases it exponentially. Usually, *cwnd* starts at some minimum window, typically one to two *MSS*, and increases by one *MSS* for each ACK received that acknowledges new data. As pointed out by [9], even with delayed acknowledgment, every other data segment elicits an ACK, and typically *cwnd* grows exponentially by a factor of 1.5. The sender enters slow start mode to quickly fill an empty "pipe" and probe equilibrium [15], in three occasions: start of a connection, after a long idle time, and after retransmission timeout, which implies severe congestion.

Once *cwnd* reaches the slow start threshold (*ssthresh*), the sender infers that the sending rate is close to the equilibrium capacity, and switches to more conservative congestion avoidance mode. In this mode, *cwnd* is increased by one *MSS* after receiving ACK's for all the packets in current *cwnd*, in effect probing the equilibrium cautiously.

Whenever a retransmission timeout occurs, the sender takes it as a signal of network congestion. It immediately set *cwnd* to initial minimum size, *ssthresh* to half of the outstanding data size, and enter into slow start mode.

#### 3.3.3.3 Fast retransmit and fast recovery

A more efficient way to detect packet loss is through duplicate ACKs when congestion is not so severe that the packets after the missing one can still be delivered and trigger duplicate ACKs. The fast retransmit algorithm uses the arrival of three duplicate ACKs as an indication that the first unacknowledged packet has been lost, and immediately retransmit this packet. Moreover, fast recovery is used right after the retransmission to allow new data packet to be sent upon receiving a duplicate ACK, which implies a packet gets delivered and left the network, until the retransmitted packet gets acknowledged. Finally, both *cwnd* and *ssthresh* are set to half the outstanding data size, and TCP enters directly into congestion avoidance mode.

A number of schemes, including *Selective Acknowledgement* (SACK) [22], *Hoe's algorithm* [13] have been proposed to improve efficiency of recovering multiple losses in a single flight of packets.

## 3.4 Existing model for TCP short connection performance

TCP performance models, which estimate the transfer time of underlying TCP flows given network performance characteristics, can be directly used to estimate the retrieval latency of Web requests. 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]. However, most of the existing TCP performance models analyze the steady-state throughput of long bulk-transfer TCP connections [11] [16] [18] [23]. The best-known model for short TCP flows experiencing losses is proposed by Neal Cardwell, et al. in [9], which gives an estimation of the flow's transfer time. Since our model is evolved from Cardwell's model (referred to as *model C* in the rest of the paper), we briefly describe it here for completeness. We will also point out its limitations to motivate our improvements to the model.

### 3.4.1 Model C

The data transfer is modeled as an initial connection establishment handshaking, followed by alternating phases of slow start and successive retransmission timeouts. This is, the transfer time

$$t = t_{handshk} + t_{RTO} + t_{xfer}$$

where $t_{handshk}$, $t_{RTO}$, $t_{xfer}$ are time spent in handshake, timeouts, and slow start respectively, and $t_{handshk} \approx RTT$.

Assume a TCP receiver that implements delayed acknowledgments sends one ACK for roughly every $b$ ($b = 2$) packets. Given the packet loss rate $p$, the number of data segments to transfer $data$, the initial slow start window for data transfer phase $w$, the round trip time $RTT$, and the average start RTO value of an RTO run $T_0$, we can compute $t_{RTO}$ and $t_{xfer}$ as follows.

#### 3.4.1.1 Estimation of $t_{RTO}$

Suppose the flow experiences $l$ losses for transfer this amount of data. Then the effective loss rate $p = \frac{l}{data+l}$ (or $l = \frac{data \cdot p}{1-p}$). Model C uses the techniques proposed in [26] to derive the probability that a particular loss will incur a retransmission timeout[4].

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

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. A sequence of $k$ RTOs occurs when there are $k-1$ consecutive losses followed by a successfully transmitted packet. Given the first loss, the probability that a RTO run includes $k$ RTOs is $p[R = k] = p^{k-1}(1-p)$ Thus, the number of RTOs in a RTO run has a geometric distribution with a mean of $\frac{1}{1-p}$, and the number of runs of RTOs, 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 as in [26]. [26] assumes that the retransmission timeout is clamped down at $64T_0$, where $T_0$ is the first RTO value of a RTO run. Thus the first six timeouts in one run have length $2^{i-1}T_0$ ($i = 1 \cdots 6$), with all subsequent timeouts having length $64T_0$. The duration of a run with $k$ timeouts is

$$L_k = \begin{cases} (2^k - 1)T_0 & \text{if } k \leq 6 \\ (63 + 64(k-6))T_0 & \text{if } k \geq 7 \end{cases}$$

Therefore, the mean duration of a RTO run is

$$t_u = T_0 \cdot \frac{1 + p + 2p^2 + 4p^3 + 8p^4 + 16p^5 + 32p^5}{1-p}$$

Thus the total time spent on RTOs is

$$t_{RTO} = u \cdot t_u$$

#### 3.4.1.2 Estimation of $t_{xfer}$

[4] While the probability model in [26] was for congestion avoidance mode, model C uses it to approximate the probability for slow start.

Since the slow start sending phase are alternating with RTO runs, the number of sending phases is $v = u + 1$. Thus the average data sent per phase is $e = \frac{data+l}{v}$. Now we need to know the time needed to send $e$ packets during a slow start phase. Consider the progress of TCP in slow start mode in terms of *rounds* whose duration is equal to one RTT. Let $cwnd_i$ be the $cwnd$ of the sender at the beginning of round $i$. Since a sender increases its $cwnd$ by one $MSS$ for each ACK received, and an ACK will be sent back for every $b$ packets, we have $cwnd_{i+1} = \left(1 + \frac{1}{b}\right) cwnd_i$. Therefore $cwnd_i$ is a geometric series with ratio $r = 1 + \frac{1}{b}$. Thus $data_i$, the number of data segments sent in rounds $1, \cdots, i$, is approximately

$$data_i = \sum_{k=1}^{i} cwnd_k = w \cdot \frac{r^i - 1}{r - 1}$$

Solving for $i$, the number of rounds needed to transfer $data_i$ segments is

$$i = \log_r \left( \frac{data_i(r-1)}{w} + 1 \right)$$

Thus the time needed per sending phase is $\log_r \left( \frac{e(r-1)}{w} + 1 \right) \cdot RTT$, and the total sending time is

$$t_{xfer} = v \cdot \log_r \left( \frac{e(r-1)}{w} + 1 \right) \cdot RTT$$

### 3.4.2 Limitations

Unfortunately, model C has the following 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 differentiate these two cases and both of them can potentially trigger a retransmission timeout. Suppose each data packet triggers an ACK packet, effective packet loss rate (i.e. timeout probability) is $p + p(1-p)$. Due to the 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 $p + p(1-p)$ as a reasonable approximation.

2. **Estimating $t_{RTO}$**

   *Clamp-down of retransmission timer.* 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. Due to Karn's algorithm [17], 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 $IT_0$s. 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 $IT_0$s is more likely to occur.

   *Measured $T_0$.* 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 3.5.2.

Figure 5 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 4(a). The curves *model C low* and *model C high* are both latency estimated using model C with same $RTT = 2 * delay$, $data = \frac{file\ size\ +\ HTTP\ header\ size}{1460}$, $w = 1$, measured $T_0$, and effective packet loss rate $p = lossrate$ and $p = lossrate + lossrate * (1 - lossrate)$, respectively. Our observations are[5]: (i) The absolute error of the estimated latency when the offered load $< 60\%$ is small; (ii) Model C underestimates due to the underestimation of the effective loss rate. Moreover, we notice that the effect of underestimated loss rate outweighs the effect of overestimated clamp-down timer; (iii) Model C with corrected effective loss rate overestimates due to the wrong modeling of clamp-down timer.

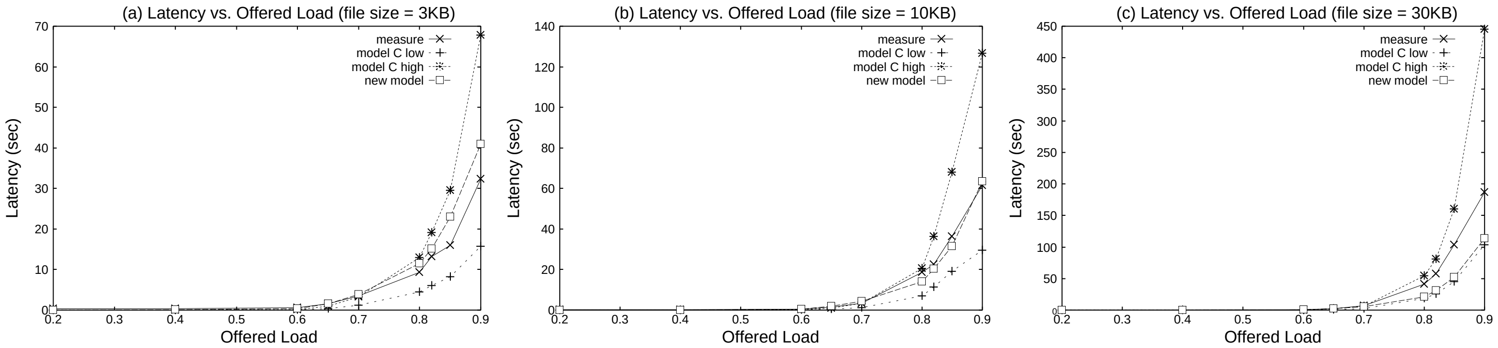


Figure 5: Offered load vs. retrieval latency: (a) file size = 3KB, (b) file size = 10KB, (c) file size = 30KB.

Figure 6 shows how well model C estimates the file transfer time when the loss rate = 0 and the delay varies. Since 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 received. In reality, ACKs may be sent back and *cwnd* may open more frequently, resulting in faster data transfer; (ii) The initial *cwnd* for data transfer phase is not necessarily one *MSS* in Linux. Unlike FreeBSD, which fixes *cwnd* to be one *MSS* during handshaking, in Linux slow start and congestion avoidance are in effect during handshaking as well as data transfer phase. It turns out that the initial *cwnd* can be two *MSS* with a considerable large probability (See Section 3.5.3). Therefore, data transfer could be faster in reality.

Figure 7 shows how well model C estimates the file transfer time when the delay = 0 and the droprate varies. The observations are similar to those obtained from Figure 5 except that the absolute error of the estimated latency is small when the loss rate $< 5\%$.

[5] 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 problematic model for clamp-down timer and the underestimation is mainly due to wrong effective loss rate.

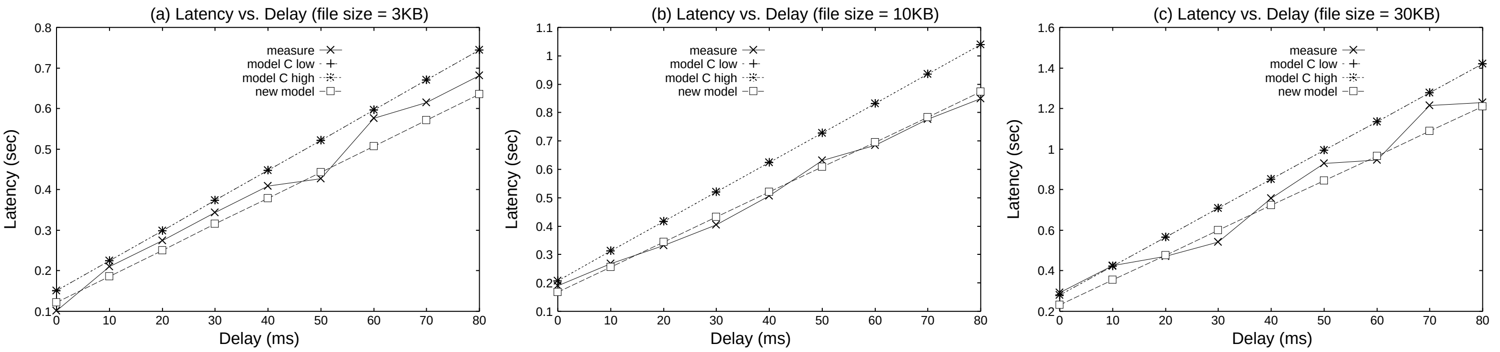


Figure 6: Packet delay vs. retrieval latency (with packet loss rate set to 0): (a) file size = 3KB, (b) file size = 10KB, (c) file size = 30KB.

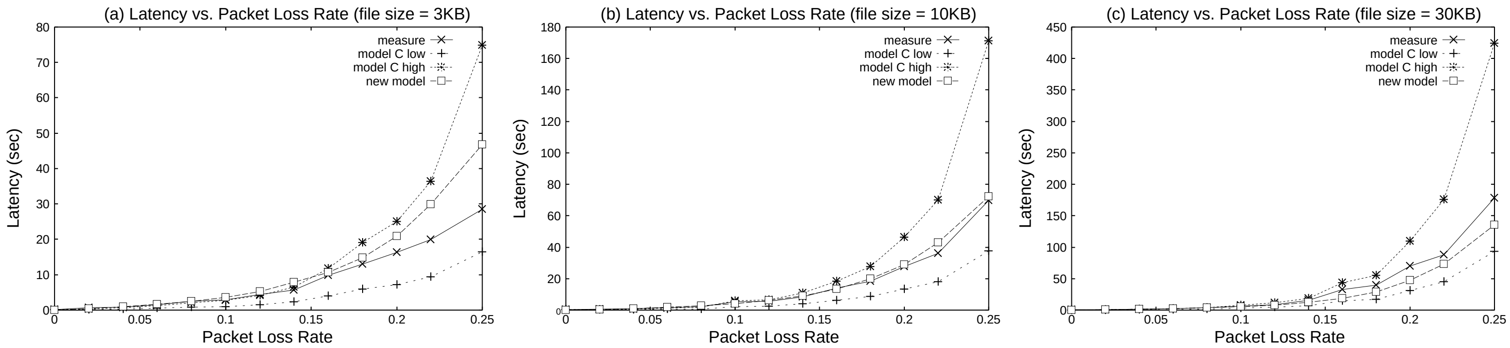


Figure 7: Packet loss rate vs. retrieval latency (with packet delay set to 0): (a) file size = 3KB, (b) file size = 10KB, (c) file size = 30KB.

We propose a new TCP 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 clamp down RTO value correctly. The computation of $t_{handshk}$ is modified to take timeouts during handshaking into account.

## 3.5 Attack problems of model C: new TCP model

As the first improvement, we use the effective packet loss rate $p = lossrate + lossrate * (1 - lossrate)$ instead of $p = lossrate$ in our new model.

### 3.5.1 Estimating $t_{RTO}$

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

The evolution of RTO value in Linux 2.0.34 implementation is depicted in Figure 8. Each node stands for a data sending (i.e. transmission and/or retransmission). The value associated with each node is the RTO value used to set the retransmission timer for the 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

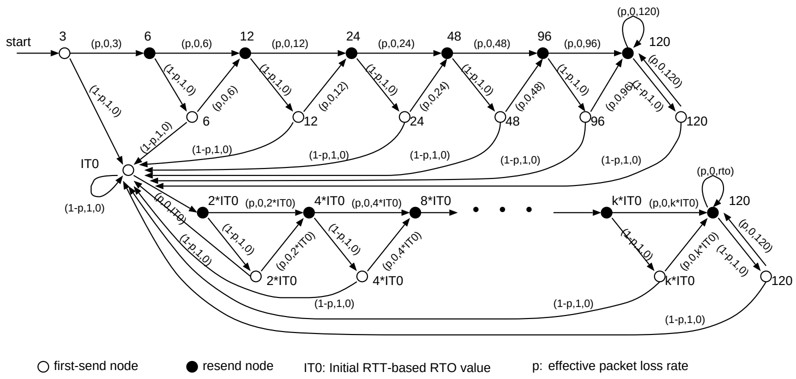


Figure 8: The Evolution of RTO value in the data transfer phase.

this edge and *time* is the time added to 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 successfully 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), the value associated with the node pointed 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:

  1. *Case 1.* 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 endnode of this edge should be an RTT-based RTO value $IT_0$;

  2. *Case 2.* 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 endnode of this edge is still *rto*.

  We can tell 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 (called *first-send* node and depicted as a circle) and it is in Case 1. If the edge is a *timeout* edge, the node is a retransmission (called *resend* node and depicted as a dot) and it is in Case 2.

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 $mdev = \frac{srtt}{2}$. Thus, the RTT-based RTO value is a constant $max(3 * srtt, 200ms) = max(6 * delay, 200ms)$ [6].

Understanding how the RTO value evolves, let's consider how to compute $t_{RTO}$.

- *Approach 1: analytical computation of the expected value*

  Ideally, we would like to analytically solve $t_{RTO}$ as the expected value of some random function. If we can compute the expected duration, $t_u$, of a single RTO run, then by using the expected number of RTO runs

[6] Linux implementation specifies the minimum RTO value is 200ms.

$u$ computed in model C, we can compute $t_{RTO}$ as $u \cdot t_u$. Denote the first timeout value for a RTO run of data packet $m$ as $T_{0m}$. From Figure 8, we have

$$T_{0m} = (1-p) \cdot IT_0 + \sum_{i=1}^{t} p^i (1-p) \cdot 2^i T_{0(m-1)} + \sum_{i=t+1}^{\infty} p^i (1-p) \cdot 120 \tag{1}$$

where $60 < 2^t T_{0(m-1)} \leq 120$, i.e. $t = \lfloor \log_2 \frac{120}{T_{0(m-1)}} \rfloor$ and $T_{01} = 3$ seconds.

Suppose that $T_{0m}$ is known. Now we compute the duration, $t_{um}$, of RTO run starting with data packet $m$. The probability that a run consists of $k$ timeouts is

$$P[R = k] = p^{k-1}(1-p)$$

The duration of a run with $k$ timeouts is

$$L_k = \begin{cases} (2^k - 1)T_{0m} & k \leq \lfloor \log_2 \frac{120}{T_{0m}} \rfloor + 1 = k_t \\ (2^{k_t} - 1)T_{0m} + 120(k - k_t) & \text{otherwise} \end{cases}$$

Thus the duration of a RTO run starting with data packet $m$ is

$$t_{um} = \sum_{k=1}^{\infty} L_k P[R = k] = T_{0m} \cdot \frac{1 - 2^{k_t} p^{k_t}}{1 - 2p} + 120 \cdot \frac{p^{k_t}}{1 - p} \tag{2}$$

Suppose that $t_{um}$ is known, the expected duration of a RTO run $t_u$ can be computed as

$$t_u = \frac{1}{data} \sum_{m=1}^{data} t_{um}$$

Unfortunately, this approach is problematic. In (1), $T_{0m}$ is not a linear function of $T_{0m-1}$. Without the knowledge of $T_{0m-1}$'s distribution, it is not obvious how to infer the relation between $E[T_{0m}]$ and $E[T_{0m-1}]$ from (1). In (2), $t_{um}$ is not a linear function of $T_{0m}$, therefore it is hard to find an analytical equation for $E[t_{um}]$ either.

- *Approach 2: simulation*

  Theoretically, once we have the evolution graph of 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 sum of *num_pkt* of edges along each path equals to *data*. Each path represents a possible way timeouts occur during data transfer. So the expected timeout time for data transfer is

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

  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 $\theta$. Using this approach, we generate sufficiently many combinations of $X_1, X_2, \cdots, X_n$ values, compute the corresponding $g(X_1, X_2, \cdots, X_n)$ values, and use the average of these values as an estimate of

$\theta$ [29]. Following this approach, to compute $t_{RTO}$ as the expected value of a function of $T_{01}, T_{02}, \cdots, T_{0,data}$, we generate sufficiently many combinations of $T_{01}, T_{02}, \cdots, T_{0,data}$ (Each combination corresponds to a path in Figure 8), use the average time spent on timeouts as a reasonable estimate for $t_{RTO}$. We implemented a small simulator (called *RTO simulator*) to perform these computations. 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.

### 3.5.2 Estimating $t_{handshk}$

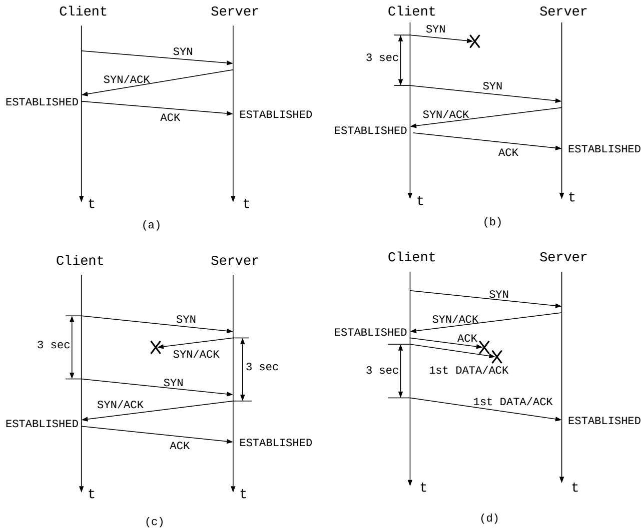


Figure 9: Four cases of the 3-way handshaking: (a) no packet loss, (b) SYN packet is lost, (c) SYN/ACK packet is lost, (d) Last ACK is lost.

In addition, we improved the estimation of the handshaking time by taking timeouts during handshaking into account. Figure 9 shows the 3-way handshaking on TCP connection for a Web retrieval session. The case when no packet is lost during this phase is depicted in Figure 9(a). The Web client initiates the connection establishment by sending SYN packet to the Web server. On receiving the SYN, the server sends its own SYN with ACK for the client's SYN piggybacked (i.e. 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 (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:

- If the SYN packet (from client to server) gets lost, it will be retransmitted and the retransmission timer keeps backing off until it is finally delivered (Figure 9(b)).
- If the SYN/ACK packet gets lost, it will be retransmitted and the retransmission timer keeps backing off until it is finally delivered (Figure 9(c)). Meanwhile the SYN packet will be timed out and retransmitted multiple times.

- If the last ACK packet gets lost, the client depends 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 9(d)).

Thus, we estimate $t_{handshk}$ as

$$t_{handshk} = ths_{xfer} + ths_{RTO}$$

where $ths_{xfer}$ is the transfer time of the packets during handshake, which is about 1.5RTT, and $ths_{RTO}$ is the time spent on timeouts.

We can view the 3-way handshake phase as the concatenation of two special data transfer phases whose $t_{RTO}$s can be efficiently estimated by simulation. The first phase is reliable transfer of the client's SYN packet, including the (re)transmissions of SYN and SYN/ACK. The second phase is either a successful transmission of the ACK packet, or a loss of the ACK packet, followed by the (re)transmissions of the first data packet of the client (without any concern about its ACK). Thus, $ths_{RTO} = ths_{RTO1} + ths_{RTO2}$, where $ths_{RTO1}$ is the timeout time for the first phase, which is $t_{RTO}$ with $p = lossrate + lossrate * (1 - lossrate)$, and $data = 1$, and $ths_{RTO2}$ is the timeout time for the second phase, which is $lossrate * t_{RTO}$ with $p = lossrate$ and $data = 1$.

### 3.5.3 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, 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 $w = (1 - p)^2 \cdot 2 + (1 - (1 - p)^2) \cdot 1$.

To sum up, by closely simulating TCP retransmission behavior, including retransmission timeout probability, RTO backoff and clamp-down, timeouts and *cwnd* change during handshaking, our model is more accurate in estimating the transfer time. In addition, by eliminating the need for measured $T_0$, our model has better serviceability.

### 3.5.4 Performance

- **Accuracy**.

  Figures 5, 6, and 7 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 6, 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 7. As a result, the new model is also more accurate in estimating the transfer time with varying load.

  Compared with the curves of measurement data, 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 5 exhibits as the combination of corresponding curves in Figure 6 and 7, hence it also overestimates for small transfer size and underestimates for large transfer size when the load is high (i.e. > 70%).

- **Speed**.

  The most time-consuming part of the new model is the computation of RTO by simulation. The RTO simulator implements a random function and needs 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.

#### 3.5.5 Discussions

Like model C, our new mode also made some simplifications to render modeling TCP performance feasible. However, these simplifications introduce some inaccuracy to the model.

- **Effect of transmit queue**

  Assuming packet loss only triggers retransmission timeout, both model C and the new model model data transfer as a handshake followed by alternating phases of slow start and RTO runs, where a RTO run is successive timeouts for one packet. Once a retransmission timeout for a packet occurs, a 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 at the same time and put into the transmit queue, 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.

  *Observation* *Suppose the transmit queue has length $> 1$. If the first packet in the transmit queue timeouts, some of the subsequent packets, if lost, will get a resend for free (i.e. the resend is not triggered by a retransmission timeout). Neither will it trigger a cwnd backoff, nor any RTO value will be added into total timeout time.*

  When sending a transmit queue of data packets out, the retransmission timer is only set for the first packet in the queue. This first packet keeps being timed out and retransmitted until an ACK for it arrives. If some subsequent packet in the queue gets lost, the ACK won't 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 its limit min(*cwnd*, *rwnd*). These packets form the updated transmit queue, and the retransmission timer will be adjusted for the first packet in this queue.

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

  Complexity arises if we want to modify our model to incorporate the effect of transmit queue. In particular, we need to:

  - Keep track of the state of transmit queue, including introducing state variables to represent the transmit queue, simulating its initialization and shift when old packets get acknowledged and new packets are sent out. Note that the manipulation of transmit queue is subject to the following restriction: the outstanding data size (approximated by number of packets in the transmit queue) at any time cannot exceed min(*cwnd*, *rwnd*) at that time.

- Keep track of *cwnd* and related information. Simulate slow start and congestion avoidance. So we know the limit of outstanding data size when we manipulate the transmit queue. This includes updating *swnd*, *ssthresh*, and *cong_count* when an ACK for new data is received, and backing off *ssthresh* and *cwnd* when a retransmission timeout occurs.
- Simulate cumulative and delayed ACK. We need to know when an ACK (as a separate packet or piggybacked) arrives in order to trigger the update of *cwnd* related information and the shift of the transmit queue.

An excerpt of server side *tcpdump* trace (Table 1) illustrates the effect of transmit queue. This episode is part of the trace when transferring 50KB file with delay = 0 and loss rate = 21%. For each packet sent/received at the server side, the table shows the time it is sent/received, the current *RTO*, *cwnd*, *ssthresh*, *cong_count*, transmit queue, and for which packet the retransmission timer is currently set. In addition, for those sends which are retransmissions, it shows how many times this packet has been retransmitted, and indicates if this resend is triggered by batch retransmission.

| data pkt index | xmit time (sec) | Nth rexmit batch(y/n) | send | recv | RTO (ms) | cwnd | ssthresh | cong count | xmit queue | pkt with timer set (RTO value set) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 12.729123 | | 36501:37961 | | | 2 | 1 | 2 | 12 | 1(*) |
| 2 | 12.729123 | | 37961:39421 | | | | | | | |
| | 12.799123 | | | ack 37961 | 390 | 3 | 1 | 0 | 234 | 2(390) |
| 3 | 12.799123 | | 39421:40881 | | | | | | | |
| 4 | 12.799123 | | 40881:42341 | | | | | | | |
| 2 | 13.119123 | 1(n) | 37961:39421 | | 780 | 1 | 1 | 0 | | |
| | 13.539123 | | | ack 39421 | (780) | 2 | 1 | 0 | 34 | 3(780) |
| 3 | 13.539123 | 1(y) | 39421:40881 | | | | | | | |
| 4 | 13.539123 | 1(y) | 40881:42341 | | | | | | | |
| 3 | 14.319123 | 2(n) | 39421:40881 | | 1560 | 1 | 1 | 0 | | 3(1560) |
| 3 | 15.879123 | 3(n) | 39421:40881 | | 3120 | 1 | 1 | 0 | | 3(3120) |
| 3 | 18.999123 | 4(n) | 39421:40881 | | 6240 | 1 | 1 | 0 | | 3(6240) |
| | 19.499123 | | | ack 40881 | (6240) | 1 | 0 | 1 | 4 | 4(6240) |
| 4 | 19.499123 | 2(y) | 40881:42341 | | | | | | | |
| | 19.999123 | | | ack 42341 | (6240) | 2 | 0 | 0 | 56 | 5(6240) |
| 5 | 19.999123 | | 42341:43801 | | | | | | | |
| 6 | 19.999123 | | 43801:45261 | | | | | | | |
| 5 | 26.239123 | 1(n) | 42341:43801 | | 12480 | 1 | 1 | 0 | | 5(12480) |

Table 1: An excerpt of server packet trace

At the start of this episode, $cwnd = 2, ssthresh = 1, cong\_count = 2$, the server sends a fresh *cwnd* of packets out with the retransmission timer set for packet 1. The transmit queue includes packets 1 and 2.

After some time, the ACK for packet 1 arrives and *cwnd*, *ssthresh*, and *cong_count* are updated accordingly. RTT estimation and RTO computation are performed with RTO value updated to 390ms. New data packets 3 and 4 are sent out. The transmit queue is shifted and now it consists of packets 2, 3, and 4. The retransmission timer for packet 2 is set to 0.390 second. When this timer goes off at time 12.729123 + 0.390 second, *cwnd*, *ssthresh*, and *cong_count* are recomputed, and packet 2 is resent with timer set to 0.780 second.

At time 13.539123 second, the ACK for packet 2 comes back. New data is acknowledged, so *cwnd*, *ssthresh*, *cong_count* are recomputed. Since we are retransmitting, no RTT estimation and RTO computation are performed. Batch retransmission is performed to resend the unacknowledged packets 3 and 4. The transmit queue now includes packets 3 and 4. The retransmission timer is doubled to 0.780 second and it is adjusted for packet 3.

Starting from 13.539123 + 0.780 = 14.319123 second, packet 3 experiences a series of retransmissions, with the timer doubling, and *cwnd*, *ssthresh* backing off. Finally at time 19.499123 second, its ACK comes back and *cwnd* etc. are recomputed. No RTT estimation and RTO recomputation need to be performed. Batch retransmission is performed and packet 4 is resent as the only unacknowledged packet left, with timer set which will go off 6240ms (the current backed-off value) after its sending time.

At time 19.999123 second, the ACK for packet 4 arrives and *cwnd* etc. get updated. No RTT estimation and RTO update need to be performed. No batch retransmission is needed because no unacknowledged packets are left. So a new *cwnd* of packets (packets 5 and 6) are sent, with timer set for packet 5 which will go off 6240ms after its sending time.

- **Delayed ACK**

  Due to delayed ACK, *srtt* in reality is an estimation of RTT plus the delay added for each acknowledgement. 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 loss rate is small.

The total inaccuracy of our model is the combination of the above three factors. 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.

# 4 Internet Telephony

In this section, we study how packet delay and loss rate, as representative network performance characteristics, affect the perceptual quality of Internet telephony application. We first describe the simulation, then analyze the measured metrics and discuss how delay and packet loss rate affect these metrics, which directly determine the user-observed quality.

## 4.1 Simulation

### 4.1.1 Application

We use the data exchange component of the Internet telephony application (provided by Cornell CS519 project group Phonedation Component 1 [24]) as application entities in our simulation testbed. The data exchange component plays the roles of both sender and receiver. As a sender, it samples audio information 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. It implements *adaptive playout* algorithm to deal with variable network delay and *Forward Error Correction* algorithm to deal with packet loss.

- **Adaptive playout**

  Due to the variability of the network, packets transmitted evenly spaced do not always arrive in order or evenly spaced at the receiver. So it is desirable for the receiver to buffer received packets for a certain amount of time before playing them out in sequence. This allows packets that arrive slightly late, or out of order, to be still in time for playout — thus creating the illusion of an in-order smooth transmission at the cost of increased *buffering delay*. Generally the larger the *buffering delay* is, more truant packets can be played out. However, the *playout delay* (*packet delay* + *buffering delay*) can not exceed 400ms, due to the fact that the excessively long delay significantly impairs human conversations [21].

  An effective way to choose the *buffering delay* is to adapt it to the delay characteristic of the network. Since the delay characteristic is not known *apriori*, usually we use an estimation of the delay based on the measurement. This is exactly the basic idea behind *adaptive playout* algorithm. The algorithm we implemented as a receiver functionality is the algorithm 2 proposed in [21] with some minor extensions[7]. The following is a sketch of the algorithm for the purpose of further discussion.

  At the receiver side, the *playout time* is computed as *send time* + *playout delay*, where *send time* is obtained from the sender's timestamp in the received packet. We estimate the optimal *playout delay* based on past measurement, and use the new estimation at the beginning of each new talkspurt[8]. Let $\hat{p}_k$ be estimates for the *playout delay* of $k$-th talkspurt, $\hat{u}$ and $\hat{v}$ be estimate for the *packet delay* and its variance during the flow, respectively. Then at the end of $(k-1)$-th talkspurt, $\hat{p}_k$ is computed as follows:

$$\hat{p}_k = \hat{u} + \beta \hat{v}$$

  To adapt to a network spike[9] efficiently, the algorithm has two modes of operation, depending on whether a spike has been detected. The two modes differ in how the estimate $\hat{u}$ is updated. In NORMAL mode, $\hat{u}$

[7] We omit the details of the implementation for simplicity. Though they slightly affect the specific data points we measured, they are irrelevant to the trend of the relationship between various parameters, which are the focus of our study.

[8] Adjusting the playout delay on a per-packet basis would lead to very jittery playout [21], so here we adjust it on a per-talkspurt basis. By talkspurt, we mean any contiguous period of audio activity between two periods of silence.

[9] A network spike is a steep rise followed by a linear, monotonic decrease back to the normal level in the packet delay [21].

and $\hat{v}$ are computed as running average of the packet delay and its variance. They are initialized when the first packet of the flow is received and updated each time a new packet $i$ arrives and its delay $d^i$ is available.

$$\begin{aligned}\hat{u}^i &= \alpha\hat{u}^{i-1} + (1-\alpha)d^i \\ \hat{v}^i &= \alpha\hat{v}^{i-1} + (1-\alpha)|\hat{u}^i - d^i|\end{aligned}$$

In SPIKE mode, $\hat{u}$ follows the rapid change of the packet delay,

$$\hat{u}^i = \hat{u}^{i-1} + (d^i - d^{i-1})$$

and $\hat{v}$ is still computed the same way as in NORMAL mode.

The switch from NORMAL mode to SPIKE mode is performed when the packet delay has a steep rise. Once in SPIKE mode, it will keep track of the sharpness of the delay change and switch back if the change is not violent any more.

The implementation uses sampling frequency 11047Hz. The sender reads 512 bytes voice data from input file or microphone every 46.3ms, packetizes it and sends it out. We use $\alpha = 0.998002$, $\beta = 4$ in the *adaptive playout* algorithm.

- **Forward Error Correction (FEC)**

  Real-time applications, such as voice and video, often lack the luxury of retransmitting lost packets. The retransmission delay, which is at least as long as the round-trip time, is larger than can be tolerated. Thus, FEC based on redundancy is attractive for real-time data streams. Moreover, previous research shows that most audio losses are isolated when the network load is low and moderate, thus open loop error control mechanisms based on FEC would be adequate to reconstruct most lost audio packets [4].

  A simple form of FEC [8] is implemented in our telephony application to handle packet loss. Each packet sent includes the current sample (in primary encoding) and redundant information of the previous sample (in secondary encoding). When such a packet arrives at the receiver, the primary sample is stored into playout buffer if it is in time for playout, and the secondary sample is stored into playout buffer if there is no corresponding primary sample scheduled to be played out and it is still in time for playout. The choice of secondary encoding is a trade-off between the additional load FEC put on the network and the quality improvement when the lost primary samples are saved by the secondary samples. In our implementation, we use PCM and ADPCM as primary and secondary encoding respectively.

### 4.1.2 Network delay and loss characteristics

The delay and loss characteristics imposed on the network are the same as in the study of Web application (Section 3.1.1). Since we use UDP in telephony application, packets may be lost on the transport protocol level due to various reasons. One typical reason that the received datagrams cannot be delivered is that the input socket buffer is full [31].

### 4.1.3 Metrics of interest

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 [33].

- *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*.

In the simulation, 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 loss distribution and delay distribution. The more even these distributions are, the lower impact low quality or lost samples will have on perceptual quality. For simplicity, we ignore this factor here. But how delay and loss distribution affect this factor, and how it affects perceptual quality need to be further studied.

In addition, we use *saved ratio* to evaluate the effectiveness of FEC, defined as

$$\frac{\textit{number of secondary samples played}}{\textit{number of lost samples} + \textit{number of late primary samples}}$$

## 4.2 Experiment 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 delay distribution) while fixing the loss rate to 0, and measure the resulting metrics[10].

Figures 10(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 10(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 (Figure 3). With the increase of delay variance, the playout delay also increases. Moreover, it changes more frequently and rapidly between different talkspurts, implying its variation also increases. This can be seen from Figure 10(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.

[10] 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.

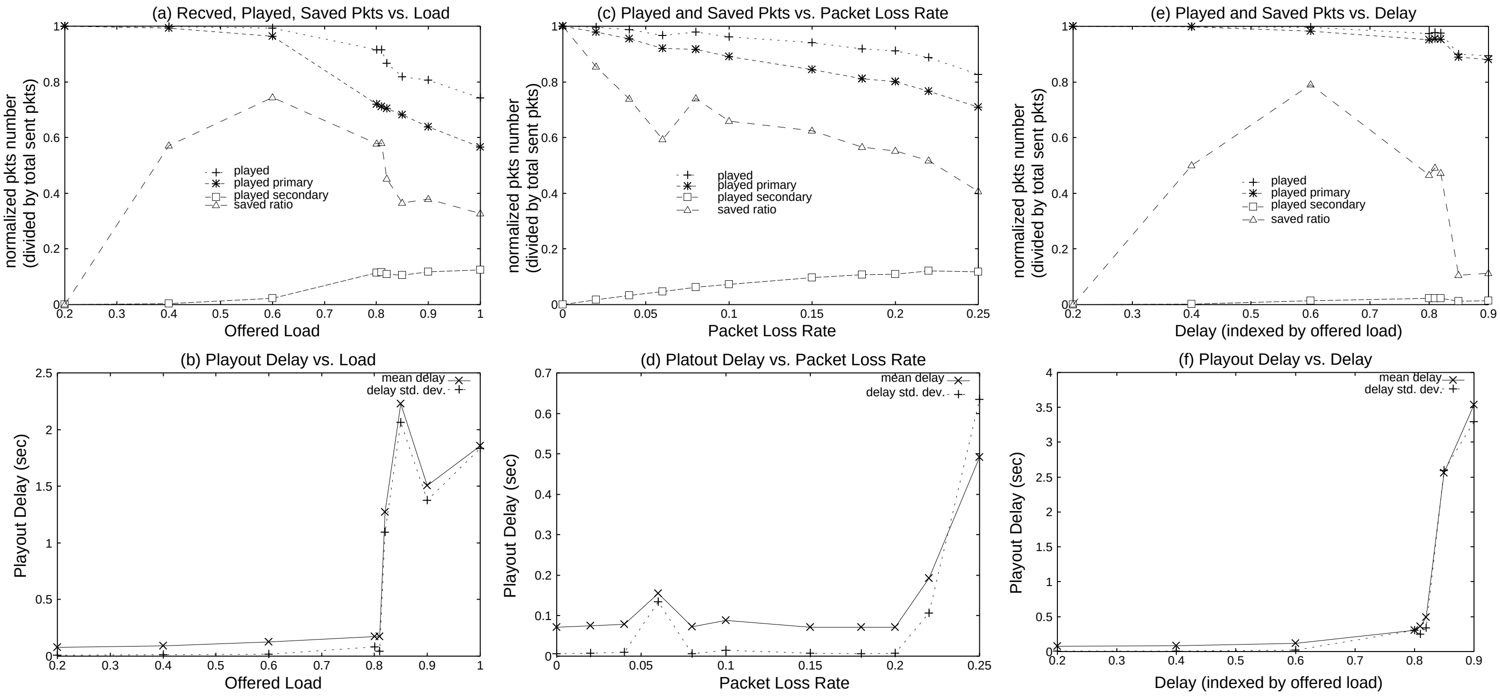


Figure 10: The experimental results of Internet telephony: (a) played and saved ratios vs. offered load, (b) playout delay vs. offered load, (c) played and saved ratios vs. packet loss rate, (d) playout delay vs. packet loss rate, (e) played and saved ratios vs. packet delay (f) playout delay vs. packet delay.

Figure 10(c) shows the effect of packet loss rate on the played ratio and the saved ratio. Almost all the received packets are played due to small delay variance. The packets saved by FEC 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[11].

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, and less sensitive to randomly distributed lost and low-quality samples, and the variation of playout delay in this case is very small due to flat packet delay.

Figure 10(e) shows the effect of delay distribution on the played ratio and the saved ratio. We fix the packet loss rate to be 0, so there is no loss introduced by the network. There are two phases in this graph:

1. *Phase 1*: offered load $\leq\sim$ 80%. The delay variance is almost 0. With small delay variance packets received 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 10(f))[12].

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

[11] 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.

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

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).

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 10(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 10(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.

# 5 Related work

The retrieval latency of a Web request can be approximated by the transfer time of the underlying TCP short connection. We base our TCP short connection performance model on model C, the best-known TCP short connection model proposed in [9]. Model C estimates the transfer time of a TCP short connection experiencing packet loss given packet delay, packet loss rate, and measured mean start RTO value of RTO runs $T_0$ *apriori*, while most of the existing TCP performance models analyze the steady-state throughput of long bulk-transfer

TCP connection with or without packet loss[11] [16] [18] [23]. Our new model improves the accuracy of model C by closely simulating TCP retransmission behavior, and enhances its serviceability by eliminating the requirement to measure $T_0$.

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 [21] [30], or FEC [7] [8] [28], 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.

# 6 Conclusions and future work

In this paper, we study the relationship between the network performance and the service quality of two representative Internet applications: the World Wide Web and the Internet telephony. We created a novel composite testbed using *Entrapid* [10] [14] (with built-in Fast Ethernet Simulation [34]), a combination of simulation and emulation which facilitates our study in LAN environment.

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, we enhanced the best known model [9] to estimate the transfer time of the underlying TCP connection of a Web retrieval. Besides enhancing the serviceability of this model, by requiring only packet delay and loss rate characteristics *apriori.* our model also improves the accuracy of this model by accounting for TCP retransmission behavior, including retransmission timeout probability, RTO value backoff and clamp-down, timeouts and *cwnd* change during handshaking. Our model can be used to map a specific delay and loss characteristics to the service quality measured by the Web retrieval latency.

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. Accordingly, the packet delay variance dominates among network characteristics which affect the user-observed service quality of Internet telephony. This can serve as a guideline for studies towards improving service quality of Internet telephony.

Although we used the packet delay and loss characteristics in LAN environment mainly because these statistics are readily available, our experimental approach is also valid for other settings, such as WAN and wireless networks, and it will be more interesting to see how network characteristics in WAN and wireless networks affect user-observed quality of applications. This is one promising direction of our future work. Moreover, for the Internet telephony, it is desirable to quantify the perceptual quality itself and the mapping between the network performance and the perceptual quality.

# References


[1] M. Allman, V. Paxson, and W. Stevens, TCP congestion control, Internet RFC 2581, April 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] J-C. Bolot and H. Crepin, Analysis and control of audio packet loss over packet-switched networks, Proc. of NOSSDAV'95.

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

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

[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] J. Hoe, Improving the start-up behavior of a congestion control scheme for TCP, Proc. of ACM Sigcomm'96.

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

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

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

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

[18] 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.

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

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

[21] 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.

[22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, TCP selective acknowledgement options, Internet RFC 2018, Oct. 1996.

[23] 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.

[24] V. Ng, K. Wagstaff, W. Weimer, and Y. Zhang, Project report for CS519, Cornell University, Dec. 1998, *http://www.cs.cornell. edu/yuzhang/519project/report/519final.htm.*

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

[26] 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.

[27] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, Modeling TCP throughput: a simple model and its empirical validation, Technical Report UMASS-CS-TR-1998-08.

[28] 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.

[29] S. M. Ross, Introduction to Probability Models (6th Edition), Academic Press, 1997.

[30] 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.

[31] W. R. Stevens, TCP/IP Illustrated, Volume 2, Addison Wesley, 1995.

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

[33] K. Toutireddy and J. Padhye, Design and simulation of a zero redundancy forward error correction technique for packtized audio transmission, Project report for CS691B, University of Massachusetts, Amherst, Dec. 1995.

[34] J. Wang and S. Keshav, Efficient and accurate Ethernet simulation, Technical Report TR99-1749, Department of Computer Science, Cornell University, May 1999.