COCOON: An Alternative Approach to End-Point Congestion Management

Yuan Gao, Guanghui He, Jennifer Hou
Department of Computer Science
University of Illinois, Urbana, IL 61801
{*yuangao,ghe,jhou*}*@cs.uiuc.edu*

Sanjoy Paul
Bell Laboratories
Lucent Technologies, Holmdel, NJ 07733
sanjoy@lucent.com

***Abstract*— In this paper, we propose an alternate endpoint congestion management scheme, called COordinated COngestion cONtrol (*COCOON*). The basic idea is to identify and group connections that may traverse the same backbone link, to enable them to share congestion information, and to coordinate among them all the congestion avoidance/control activities. The size of a *COCOON* group can be dynamically adjusted so as to magnify the benefits of end-host congestion management. *COCOON* also allows a new connection to commerce with a congestion window that is large enough to catch up with other connections while not inducing congestion. Finally, *COCOON* takes into account non-responsive UDP connections and "bundles" them into a virtual connection that is subject to TCP-like congestion control.**

Both our *ns-2* simulation and FreeBSD empirical studies indicate that as compared to other existing schemes such as *TCP-Reno, TCP-Int, Congestion Manager, HTTP 1.0* and *HTTP 1.1, COCOON* significantly reduces the packet loss ratio of concurrent connections, while sustaining the throughput comparable to the best scheme. This, coupled with the fact that *COCOON* requires only minor modification at server hosts, suggests that *COCOON* can be readily deployed over the Internet.

***Index Terms*— End-point congestion management, TCP, World Wide Web, HTTP.**

The work reported in this paper was supported in part by NSF under Grant No. ANI-0073725 and No. ANI-9903160. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

I. Introduction

As reported in [1], TCP traffic constitutes the majority — as high as 95% — of the Internet traffic, and World Wide Web (WWW) constitutes approximately 65–75% of the TCP traffic today. This is because web servers and clients communicate with each other using the Hypertext Transfer Protocol(HTTP/1.0 or HTTP/1.1) [2], [3] which uses TCP as the underlying reliable transport protocol [4]. Consequently, the Web traffic has become the dominant component of the Internet backbone traffic. This observation is corroborated by the statement in [5] that the Web traffic constitutes 50% of the bytes traversing busy backbone links.

The rapid growth of WWW traffic has caused significant characteristic changes in the Internet data traffic. This is because the characteristics of WWW traffic are significantly different from those of other traditional TCP applications (such as Telnet and FTP) in the following three aspects. First, a web page typically contains several embedded components (e.g. images). In order to reduce the latency perceived by users, a Web browser often launches multiple, simultaneous TCP connections to a given server to obtain the embedded objects of a web page. Second, the average size of a web page and its embedded components is usually small, and hence the TCP connections initiated to retrieve a web page and its embedded components are usually short-lived. Third, web surfing is inherently an interactive activity, and hence the response time is an important performance criterion from the user's perspective.

The above distinct characteristics give rise to several issues which the current TCP does not adequately deal with. First, as most TCP connections initiated by HTTP are short-lived and do not have enough time to probe, and fully utilize, the available network bandwidth under the *slow-start* phase, they usually experience unnecessarily long response times. This is due to the fact that the window size of a short-lived connection may not reach the congestion avoidance threshold (*ssthresh*) before the connection completes its transmission. (In comparison, for a long-lived connection, e.g., a *FTP* connection that transfers a large file, this *slow-start* period is short as compared to the entire duration of the connection, and the average window size over the entire duration is reasonably large.) Second, use of multiple, concurrent connections for web page retrieval may lead to blind competition among these connections (and hence the increased likelihood of traffic burstiness).

HTTP/1.1 [3] aims to alleviate the above problems by enabling a web client/server to transfer multiple components sequentially over one persistent TCP connection. The persistent connection approach avoids blind competition among concurrent connections, and allows a new connection to start with a larger congestion window (and hence the small window size in the *slow-start* phase does not impact the response time so dramatically). However, as reported in [7], only 21% of all (1.35 billion) requests to FIFA'98 websites used HTTP/1.1. A similar trend was also reported in [9] for requests made to the NASA's Glenn Research Center website over a 1 1/2 year time period. Moreover, use of HTTP/1.1 only solves the aforementioned issues in the case that a single client requests multiple components from the same server. Connections from a server destined for the same subnet (e.g., employees from the same corporate may connect to the same website to quote/trade stocks simultaneously) and/or connections established by applications other than WWW may still compete for network resources with one another. What is needed is an endpoint congestion management (ECM) scheme that is not application-specific.

In this paper, we propose an ECM scheme, called COordinated COngestion cONtrol (*COCOON*). The basic idea is to identify and group connections that are destined for the same destination host/subnet and coordinate among them all the congestion avoidance/control operations (Fig. 1). The rationale behind grouping connections destined for the same subnet is grounded on several characteristic studies of WWW client-based traces (e.g., [6], [8]): clients on the same subnet usually do share common interest, and it is highly likely that more than one clients from the same subnet visit some popular website at the same time. (That is, the previous example of employees from the same corporate connecting to the same financial website is common.) We also devise a group management approach to dynamically adjust the granularity of *COCOON* groups so as to magnify the advantage of control information sharing and congestion control coordination. *COCOON* also regulates non-responsive UDP connections and bundle those that are destined for the same destination host/subnet into a virtual connection which is then subject to TCP-like congestion control. *COCOON* requires only minor modification at server hosts, is totally transparent to client hosts, and hence can be incrementally deployed over the Internet.

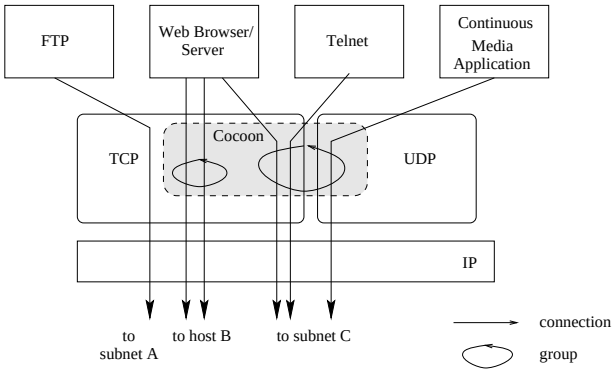


Fig. 1. *COCOON* in the Internet protocol stack.

Several approaches have been proposed in the realm of ECM, among which T/TCP [14], [15], [16], *TCP-Int* [19], *Ensemble-TCP* [18], *TCP Fast Start* [20], [21], and most recently *Congestion Manager* (CM) [23] may have received the most attention. We will provide a more detailed summary of these approaches in Section V. The work that comes closest to ours is CM. Conceptually, a CM sender

ensembles concurrent unicast flows with the same destination host into a single flow. All the flows are subject to a single congestion window governed by a window-based Additive-increase/multiplicative-decrease(AIMD) congestion control algorithm. In particular, packet loss from any of the flows leads to multiplicative decrease of the congestion window. The major differences between CM and *COCOON*, however, lie in (i) *COCOON* coordinates congestion management not only among connections destined for the same host but also among connections destined for the same *subnet*. This maximizes the advantage of congestion information sharing; (ii) all concurrent connections are *not* subject to the same congestion window, but are instead governed by their individual congestion windows. What is being shared is the congestion information. As a result, a single packet loss will not significantly reduce the instantaneous throughput attainable by *all* the flows as in CM. This also eliminates the need to employ traffic shapers or schedulers to regulate/schedule packet transmission among flows, as required in CM; (iii) *COCOON* does not introduce additional control protocol/messages to elicit feedback information; and (iv) *COCOON* requires modification only at senders (which are usually servers) and is transparent to receivers. Both our *ns-2* simulation and FreeBSD empirical studies over the Internet indicate that as compared to *TCP-Reno*, *TCP-Int*, *Ensemble-TCP*, CM, HTTP/1.0, and HTTP/1.1, *COCOON* significantly reduces the packet loss ratio, while sustaining the throughput comparable to the best scheme.

The rest of this paper is organized as follows. We present *COCOON* in Section II, and elaborate on the grouping issues (i.e., how groups are dynamically merged or split, and to which group a new connection should join) in Section III. We analyze, with the use of Markovian modeling, the cost of group management in the worst case in Section IV. We summarize the related work in Section V and evaluate *COCOON* against other schemes via *ns-2* simulation and Internet experiments in Section VI. Finally, we conclude the paper in Section VII.

II. Coordinated Congestion Control

COCOON resides at a sender host, and identifies and groups concurrent TCP and UDP connections that are destined to the same destination host or subnet. The rationale behind this design is that these connections are likely to be routed along some common backbone links, and are subject to similar congestion characteristics and round trip times (*RTT*s). Hence, they should coordinate on their congestion control activities. *COCOON* enables these connections to share the congestion information so that (i) they can, rather than competing blindly, cooperate with one another in congestion control; (ii) new connections may leverage the shared congestion information and commence with a reasonably large congestion window; and (iii) UDP connections can be regulated so as not to deprive well-behaved TCP connections of their fair resource share.

The granularity of a connection group that is subject to the same "fate" is dynamically adjusted. The finest grain, called an *atom*, is the bundle of all TCP/UDP connections destined for the *same destination host*. Connections whose destination hosts have the same network prefix (e.g., the IP addresses of destination hosts AND'ed with a subnet mask) can also form a group, if they are "observed" to share certain backbone links. Under this case, a group may consist of several atoms. We will discuss the issue of group management, in particular, how connections destined for the same subnet are "observed" to share some backbone links in Section III.

A. Congestion Information Shared in a COCOON Group

Under proper group management, all the connections in a group are routed along the backbone links and hence are subject to the same *RTT* (plus/minus a minor variation of delay incurred on different subnets). As a result, they are instrumented to share the information of *RTT* and *RTT* variance. The way in which a group estimates its group *RTT* and group *RTT* variance is the same as that in which a TCP connection estimates these parameters [10]. Specifically, when a connection in a group receives a non-duplicate acknowledgment, the group estimates its group *RTT* (*RTT* variance) as the weighted average of the previous estimate and the new measured value. If connections that traverse the

/* Upon receipt of 3 duplicate ACKs for the packet with sequence number k, connection C_i performs the following operation */
If ($k >$ *water_mark*) {
    $cwnd_i \leftarrow \frac{1}{2}$ *FlightSize*$_i$;
    inform connection C_j in the same group;
}

/* Upon being notified by connection C_i in the same group of its packet loss, C_j performs the following operation */
If *FlightSize*$_j \geq$ *FlightSize*$_i$ {
    $cwnd_j \leftarrow \max(\frac{1}{2}$*FlightSize*$_j$, *FlightSize*$_i)$;
    ssthresh$_j \leftarrow cwnd_j$;
    /* congestion avoidance phase commences */
}
else if *FlightSize*$_j \geq \frac{1}{2}$ *FlightSize*$_i$ {
    $cwnd_j \leftarrow$ *FlightSize*$_j$;
    ssthresh$_j \leftarrow \frac{1}{2}$*FlightSize*$_i$;
    /* congestion avoidance phase commences */
}
else
    ssthresh$_j \leftarrow \min(\frac{1}{2}$*FlightSize*$_i$, *ssthresh*$_j)$;
/* update water mark to avoid repeated reduction of the window size within one round trip time. */
water_mark $\leftarrow$ the largest sequence number of C_j's current outstanding data;

Fig. 2. The window adjustment procedure in the case that a connection incurs packet loss.

same backbone links are indeed grouped into the same group, the estimate is more accurate, because approximately n times more samples are used in the estimation, where n is the number of connections in the group.

The group *RTT* and *RTT* variance are used to set up retransmission timers for all the connections in the group, in the same way in which a TCP connection sets up its retransmission timer. Note that the information shared is the information a TCP connection typically uses for congestion control, but not the user information, and hence security/privacy is not compromised.

B. Coordinated Congestion Control for TCP Connections in a COCOON *Group*

In addition to sharing congestion information, all the connections in a group coordinate on congestion control activities as follows:

When a connection detects congestion:: When a connection C_i receives 3 duplicate acknowledgments (which is taken as the signal for packet loss), the congestion window (*cwnd*$_i$) of C_i is reduced to half of its outstanding data size, *FlightSize*$_i$, as described in [11] (i.e., *cwnd*$_i \leftarrow \frac{1}{2}$ *FlightSize*$_i$). Also, since packet loss may be a result of other concurrent connections aggressively probing the available bandwidth, the other connections in the same group should also adjust their congestion windows. In *COCOON*, the procedure used to adjust the congestion windows of these connections is given in Fig. 2 and discussed below:

(1) If the outstanding data size, *FlightSize*$_j$, of a connection C_j is greater than or equal to that of connection C_i, C_j may potentially be responsible for the congestion, and hence its congestion window size, *cwnd*$_j$, is reduced to $\max\{\frac{1}{2}$*FlightSize*$_j$, *FlightSize*$_i\}$. That is, if *FlightSize*$_j \geq 2\times$ *FlightSize*$_i$, the *cwnd*$_j$ is adjusted as if C_j experienced packet loss; otherwise, *cwnd*$_j$ is set to *FlightSize*$_i$. In the latter case, the reduction in the congestion window is not as aggressive as *cwnd*$_j \leftarrow \frac{1}{2}$ *FlightSize*$_j$ so that C_j does not experience significant throughput fluctuation. The slow start threshold, *ssthresh*$_j$, is also set to the same value of *cwnd*$_j$ so as to enforce C_j to enter the congestion avoidance phase.

(2) If *FlightSize*$_j$, of a connection C_j falls in $[\frac{1}{2}$ *FlightSize*$_i$, *FlightSize*$_i)$, then *cwnd*$_j$ is set to *FlightSize*$_j$, implying that C_j is not allowed to send any data until receipt of a new acknowledgment. Moreover, *ssthresh*$_j$ is set to $\frac{1}{2}$ *FlightSize*$_i$. Since *cwnd*$_j \geq$ *ssthresh*$_j$, the congestion avoidance phase commences.

(3) If *FlightSize*$_j$ of a connection C_j is less than $\frac{1}{2}$ *FlightSize*$_i$, then *cwnd*$_j$ is not adjusted because C_j does not grasp the bandwidth as aggressively as C_i and should not be penalized. However, *ssthresh*$_j$ is set to $\min\{$*ssthresh*$_j, \frac{1}{2}$*FlightSize*$_i\}$, so that C_j

will commence the congestion avoidance phase (and behaves less aggressively) when its congestion window reaches $\frac{1}{2}$ *FlightSize*$_i$.

When a connection incurs timeout:: When a congestion C_i incurs timeout, it commences the slow-start phase. The connections in the same group whose outstanding data size is larger than or equal to $FlightSize_i$ reduce their congestion windows to half of their outstanding data size, while those whose outstanding data size is less than $FlightSize_i$ do not reduce their congestion windows, but instead set *ssthresh*$_j \leftarrow \frac{1}{2}FlightSize_j$.

When multiple packet losses or timeouts occur within one round trip time:: To prevent the congestion window from being reduced multiple times within one *RTT*, each TCP connection keeps a water mark which is set to the largest sequence number of its outstanding data when the congestion window was last adjusted. When a connection incurs packet loss or timeout, it compares the sequence number of the lost packet or the packet that incurs the timeout against its water mark. Only when the former is larger than the latter will the connection, along with the other connections in the same group, adjusts its congestion window.

C. Window Set Up for New Connections in a COCOON *Group*

To allow a new connection to start with a larger congestion window to reduce latency while not inducing congestion, we propose the following window setup procedure in *COCOON:* When a new TCP connection, C_k, is established and added to a group, it chooses as its *reference connection* the TCP connection, C_r, with the smallest congestion window among all the *active* TCP connections in the group. (By "active," we mean connections with data to send.) C_k then sets its congestion window to $cwnd_k \leftarrow \frac{1}{2}cwnd_r$ and its slow start threshold to *ssthresh*$_k \leftarrow \min\{cwnd_r,$ *ssthresh*$_r\}$. In the case that there is no active TCP connection at the time when C_k joins a group, C_k follows the TCP slow start procedure and starts with $cwnd_k = 1$.

Note that by setting $cwnd_k \leftarrow \frac{1}{2}cwnd_r$, the new connection will catch up with the reference connection in one round trip time and will not suffer from the effect of a small window size in the slow start phase. On the other hand, by setting *ssthresh*$_k \leftarrow \min\{cwnd_r,$ *ssthresh*$_r\}$, the new connection always starts in the slow start phase, but if the reference connection is itself in the congestion avoidance phase ($cwnd_r \geq$ *ssthresh*$_r$), the new connection will enter the congestion avoidance phase in one round trip time, thus not behaving too aggressively.

D. Coordinated Congestion Control for UDP Connections in a COCOON *Group*

If TCP and UDP connections co-exist in a group, to prevent non-responsive UDP connections from depriving TCP connections of their fair share of network resources, *COCOON* bundles all the UDP connections in a group as a *virtual* connection. The virtual connection is then regulated by a *virtual* congestion window, $cwnd_v$. The rules for adjusting $cwnd_v$ and *FlightSize*$_v$ of the virtual connection are as follows:

(1) The congestion window, $cwnd_v$, of the virtual connection is set to the average congestion window (averaged over all the TCP connections in the group) times the number of UDP connections. If at the time when the virtual connection is established, no TCP connection exists in the group, $cwnd_v$ is set to a default value. The outstanding data size, *FlightSize*$_v$, is initialized to 0.

(2) When a virtual connection is established (i.e., some UDP connection(s) are established in a group), it is given the congestion window $cwnd_v$ calculated above. UDP connection(s) in a group are allowed to send packets in a round-robin fashion (for the purpose of fairness) only if *FlightSize*$_v$ is less than or equal to $cwnd_v$. When a UDP packet is sent, *FlightSize*$_v$ is increased by the amount of data bytes sent. When a non-duplicate acknowledgment is received by a TCP connection in the group, $cwnd_v$ is updated (after the TCP connection's $cwnd$ is updated), and *FlightSize*$_v$ is reduced by the amount of TCP data bytes acknowledged. This, in

some sense, regulates the aggressiveness of UDP connections to that of TCP connections.

(3) When a TCP connection incurs packet loss and reduces its congestion window, the virtual connection updates its $cwnd_v$ accordingly. If after the update, $FlightSize_v \geq cwnd_v$, the virtual connection freezes its transmission until enough non-duplicate acknowledgments are received by TCP connections in the same group and the condition $cwnd_v > FlightSize_v$ holds again.

(4) When a TCP connection times out, the virtual connection immediately freezes its transmission, no matter whether or not $FlightSize_v \geq cwnd_v$. The virtual connection only resumes transmission (with a updated $cwnd_v$), when a new acknowledgment is received by the TCP connection that incurs timeout.

(5) If all TCP connections in the group are either inactive (i.e., idle without packets to send) or closed, no acknowledgments will be received within the group. Under this case, $cwnd_v$ and $FlightSize_v$ remain unchanged and the virtual connection is allowed to send at the rate $cwnd_v/group_RTT$, until one of the existing TCP connections becomes active again or a new TCP connection is established. In the period, UDP connections may not respond to network dynamics, due to the lack of up-to-date control messages.[1]

Note that, only when UDP connections co-exist with some TCP connections that are destined for same destinations, UDP connections are subject to control by the virtual connection, otherwise, they are not controlled.

[1]Alternatively, *COCOON* may instrument UDP connections to respond to network dynamics by artificially creating a TCP connection with the same destination as UDP connections, and having the TCP connection periodically send one-byte date packets to probe the network status on the behalf of the UDP connection. This is a subject of future research.

III. Group Management in *COCOON*

Determination of an appropriate group size so as to coordinate congestion activities among group members that are truly subject to the same backbone links is not a trivial task. On the one hand, if connections that do not share the same backbone links are grouped together, packet loss incurred in one connection may cause other irrelevant connections to reduce their sending rates, thus impairing attainable throughput. On the other hand, if connections that indeed share the same (bottleneck) backbone links are not grouped together, the advantage of ECM is not fully magnified. In what follows, we propose a two-step group management approach in Section III-A, and analyze its overhead in Section IV.

A. *Establishment of* COCOON *Groups*

The first step is to group connections with destination hosts on the same subnet. This could be done by ANDing the destination host address with an appropriate subnet mask (e.g., $255.255.0.0$ for Class B addresses and $255.255.255.0$ for Class C addresses) and grouping connections with the same network prefix. By changing the subnet mask, we can change the granularity of groups. But because of the use of Classless Inter-Domain Routing (CIDR) in Internet, using $255.255.255.0$ as subnet mask results in less error in grouping connections.

In the second step, we further decompose groups that consist of destination hosts on the same subnet into smaller groups. This is based on the observation that for connections destined for the same subnet, if they indeed share some common bottleneck backbone link(s), they usually experience similar queuing and packet loss characteristics and are thus subject to similar average *RTT*s and *RTT* variances. Hence, we further classify connections whose *RTT*s are apart from one another by $c \cdot RTT_variance$ into different groups, where c is a constant. Specifically, for each coarse-grain group formed in the first step, we apply the following algorithm:

Step 1. The connections are sorted in the non-descending order of their RTTs.

Step 2. Let R and V be the *RTT* and the variance of *RTT* of the first connection C_1 in the

sorted list.

Pick up the next connection, C_i, in the list. Let R' and V' be the *RTT* and the *RTT* variance of connection C_i. If

$$R' \leq R + cV \quad \text{and} \quad R \geq R' - cV', \quad (1)$$

then tag C_i and go to **Step 2**; otherwise go to **Step 3**.

Step 3. The connections C_1 through C_{i-1} should form a group. Remove C_1 through C_{i-1} from the sorted list. If the list is not empty, go back to **Step 2**; otherwise, terminate the algorithm.

Several comments are in order: first, the time complexity of the above algorithm is $O(n \cdot logn)$, where n is the number of connections in the group. Second, the parameter c is tunable and controls to what extent connections of different *RTT*s can be grouped together. In the simulation results to be presented in Section VI-A, we set c to be in the range of $[1, 2)$.

B. Dynamic Group Merging and Splitting

As connections may be established and terminated dynamically, the number of groups as well as the size of each group may change over time. To ease group management and to magnify the benefit of ECM, we propose to dynamically adjust/control the above two parameters. Specifically, when the number of groups exceeds a pre-determined upper bound, M, we merge smaller groups with destination hosts *on the same subnet* into a larger one (to magnify the advantage of ECM). This is done by (i) selecting groups with destination hosts on the same subnet, and (ii) applying the algorithm described above, but with each group treated as a connection (i.e., the term "connection" is replaced with "group" and the term "*RTT*" with "group *RTT*" throughout the algorithm). Also, to allow group merging, the parameter c is selected to be in $[2, 3)$. The group *RTT* (group *RTT* variance) of each newly merged group is the arithmetic (geometric) average of the group *RTT*s (group *RTT* variances) of individual groups.

When the size of a group exceeds a pre-determined upper bound, N (perhaps because of establishment of new connections), we apply the above algorithm (with $c \in [1, 2)$) to decompose it into multiple, smaller groups. If after application of the above algorithm, the size of a group is still larger than N, we can switch the first '0' to '1' in the subnet mask to divide the group into two subgroups. The group *RTT* of each newly split group is initially set to that of the old group and will be updated as acknowledgments arrive within that group.

C. Group Assignment of New Connections

When a new connection C_ℓ is established, it is assigned to a group with connections destined for the same subnet (if such a group exists). Because of the ways in which groups are merged and split, there may be multiple groups with the same network prefix as C_ℓ, and the longest matching criteria is used to decide which group C_ℓ should join.[2] In case that such a group does not exist, C_ℓ itself constitutes a new atom.

IV. Cost Analysis of Group Management

To analyze the cost of group management (in terms of the fraction of system time required to perform group management), we derive a continuous-time Markov chain model. The model serves to quantify the overhead incurred in group management with respect to the connection setup rates, connection durations, the maximum size, N, of a group, and the maximum number, M, of groups.

A. Modeling with a Continuous-Time Markov Chain

For ease of analysis, the time is normalized to the time it takes to execute the group merging/splitting algorithm (for N connections) given in Section III-A (which is typically in the range of $\approx$ 10 milliseconds). We characterize the view of a *COCOON* group (called the *tagged* group) toward the system

[2] Instead of linearly comparing the network prefix of all the existing groups, we use a simple two-stage search method. In the first stage, C_ℓ's designation IP address is ANDed with an appropriate subnet mask and this value is used as a hash key into a hash table. All the groups with the same hash key is linked together. In the second stage, the linked list is then further searched to find the longest match.

in the state (n, m), where n is the number of connections in the tagged group, and m is the current number of groups the host has. We lump all the states (n, m) with $n \geq N+1$ into a single state $(N+1, m)$, as the group splitting operation will be invoked on all such states. Similarly, all the states (n, m) with $m \geq M+1$ are lumped into a single state $(n, M+1)$, as the group merging operations will be invoked on all such states. Furthermore, we make the following assumptions:

- A1. Connections are established according to a Poisson process with rate λ, and the distribution of connection duration is exponential with mean $1/\mu$.
- A2. The number of destination subnets to which connections of a host may connect is no more than K, and it is equally likely for a TCP connection to be connected to any destination subnet.
- A3. The size of the $(m-1)$ groups (other than the tagged group) in the state (n, m) is uniformly distributed over $[1, N]$. Hence, probabilistically $\frac{m-1}{N}$ groups contain i connections, $1 \leq i \leq N$.
- A4. When the number of connections in the tagged group exceeds N, the tagged group is split into two subgroups with size i and $N-i$, respectively, with probability $\frac{1}{N}$, $1 \leq i \leq N$. The subgroup with size i is selected as the new tagged group.
- A5. When the current number of groups exceeds M, only a (randomly selected) pair of groups is merged.[3] Given a current state $(n, M+1)$, the probability that the tagged group is selected for merging is thus $\frac{M}{C_2^{M+1}} = \frac{2}{M+1}$. Furthermore, if the tagged group is selected, then under **A2**, the probability that the size of the newly merged group is $n+i$ is $\frac{M}{N}$ for $1 \leq i \leq N$.

With the above assumptions, the system evolution can be modeled as a continuous-time Markov chain $\{X(t), t \geq 0\}$. Most of the above assumptions are made based on the notion of randomness, and may not necessarily represent the subnet and group distributions in the real world. However, due to the fact that appropriate statistical models that characterize these distributions of interest are lacking, the Markovian model serves to give, under the most random case, estimates in the overhead incurred in group management *under an extremely wide spectrum of workloads*, with the ratio of connection duration to connection inter-arrival time varying from 0.01 to 500. The state transition probability, $P_{(n,m)\to(n',m')}$, for the Markov chain is shown in Table. I. The state transition diagram is shown in Fig. 3.

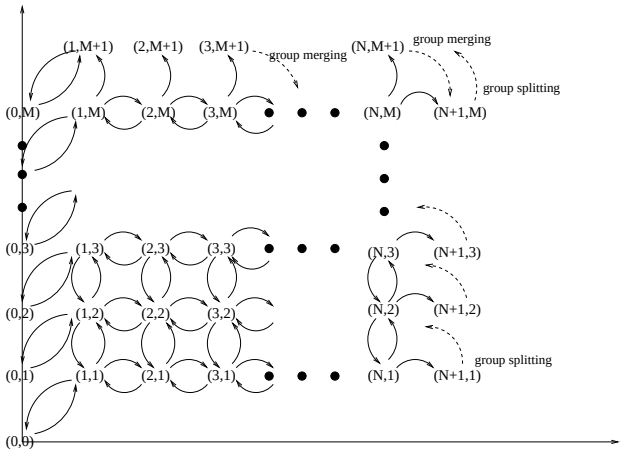


Fig. 3. The state transition diagram that models the view of the tagged group toward the system.

B. Derivation

Let $S_N \triangleq \{(N+1, m) \mid 1 \leq m \leq M+1\}$ denote the set of states that trigger the group splitting operation, and $S_M \triangleq \{(n, M+1) \mid 1 \leq n \leq N\}$ the set of states that trigger the group merging operation. Also, let T_N denote the time the system first enters a state $\in S_N$, given that the system starts in a state $\notin S_N$ (i.e., $T_N = \min\{t : X(t) \in S_N\}$), and T_M the time the system first enters a state $\in S_M$, given an initial state $\notin S_M$. By use of the randomization technique [26], [27] outlined in Appendix, we can derive $E(T_N)$ and $E(T_M)$. In what follows, we derive $E(T_N)$. As $E(T_M)$ can be similarly derived, its derivation is omitted.

As described in Appendix, we decompose the continuous-time Markov chain $\{X(t), t \geq 0\}$, into two component processes: a discrete-time Markov

[3]Note that this assumption errs on the pessimistic side.

Event that triggers the transition	(n', m')	Trans. rate
The first connection is established in the tagged group	$(n+1, m+1)$, $n = 0$, and $0 \le m \le M$	$\lambda \cdot \frac{1}{K}$
A connection is established in the tagged group	$(n+1, m)$, $1 \le n \le N$, and $1 \le m \le M$	$\lambda \cdot \frac{1}{K}$
A connection is established and a new group is formed	$(n, m+1)$, $1 \le n \le N$, and $1 \le m \le M$	$\lambda \cdot (1 - \frac{m}{K})$
A connection in the tagged group terminates	$(n-1, m)$, $2 \le n \le N$, and $1 \le m \le M$	$n \cdot \mu$
The only connection in the tagged group terminates	$(n-1, m-1)$, $n = 1$, and $1 \le m \le M$	μ
The only connection in a group (other than the tagged group) terminates	$(n, m-1)$, $1 \le n \le N$, and $2 \le m \le M$	$\frac{m-1}{N} \cdot \mu$
The tagged group is split	$(i, m+1)$ for $1 \le i \le N$, where $n = N+1$, and $1 \le m \le M$	$\frac{1}{N}$
A pair of groups is merged (the tagged group is not selected)	$(n, m-1)$, $m = M+1$, $1 \le n \le N$	$\frac{M-1}{M+1}$
A pair of groups is merged (the tagged group is selected)	$(n+i, m-1)$ for $1 \le i \le N-n$, where $m = M+1$, $1 \le n \le N$	$\frac{2}{M+1} \cdot \frac{M}{N}$
	$(N+1, m-1)^{\ddagger}$, where $m = M+1$, $1 \le n \le N$	$\frac{2}{M+1} \cdot \frac{M}{N} \cdot n$

‡ Recall that the states $(n + (N-n+1), M)$, $(n + (N-n+2), M)$, …, $(n+N, M)$ are all lumped into state $(N+1, M)$.

TABLE I

STATE TRANSITION PROBABILITY $P_{(n,m)\to(n',m')}$

chain $\{Y_n, n = 0, 1, \ldots\}$ and a Poisson process $\{N(t), t \ge 0\}$. Let $p(n,k)$, $0 \le k \le n+1$, denote the probability that the underlying discrete-time Markov chain (obtained after randomization) visits states $\notin S_N$ k times, given n state changes, and $P_s(t)$ denote the probability that the group splitting operation is triggered within t, given an initial state $\notin S_N$. Then, $p(n, n+1)$ is the probability that the underlying Markov chain always stays in states $\notin S_N$ during these n state changes. Consequently, the probability that no group splitting operation is triggered, $1 - P_s(t)$, is the probability that the underlying Markov chain always stays in states $\notin S_N$ regardless of the number of state changes in $[0, t]$, i.e.,

$$\begin{aligned} 1 - P_s(t) &= \sum_{n=0}^{\infty} p(n, n+1) \cdot P(n \text{ state changes}) \\ &= \sum_{n=0}^{\infty} p(n, n+1) \cdot \frac{e^{-\Lambda t}(\Lambda t)^n}{n!}, \end{aligned}$$

where Λ is the rate of the underlying Poisson process obtained after randomization. The error resulting from the truncation of the infinite sum in Eq. (2) can be easily bounded. Specifically, let R_m denote the error resulting from truncating Eq. (2) to m steps, then

$$\begin{aligned} R_m &= \sum_{n=m+1}^{\infty} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} \cdot p(n, n+1) \\ &\le \sum_{n=m+1}^{\infty} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} = 1 - \sum_{n=0}^{m} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!}, \end{aligned}$$

where $\le$ in the above expression results from the inequality $p(n, n+1) \le 1$. m can be evaluated *a priori* for a given error tolerance.

Now we calculate $p(n,k)$. Let $p(n, k, a_i)$ be the probability that the underlying Markov chain visits states $\notin S_N$ k times out of n steps and let a_i be the state visited after the last transition. Then, we have

$$p(n,k,a_i) = \begin{cases} \sum_{j=1}^{|S|} p(n-1, k-1, a_j) \cdot P_{ji} & \text{if } a_i \notin S_N, \\ \sum_{j=1}^{|S|} p(n-1, k, a_j) \cdot P_{ji} & \text{if } a_i \in S_N, \end{cases}$$

where P_{ji} is the transition probability from state a_j to state a_i in the underlying discrete-time Markov chain $\{Y_n, n = 0, 1, ...\}$ (Eq. 6 in the Appendix). The initial conditions are given by

$$p(0, 1, a_i) = \begin{cases} \pi_i(0), & \text{if } a_i \notin S_N, \\ 0, & \text{otherwise,} \end{cases}$$

where $\pi_i(0) = \phi_i(0)$ is the transient probability of the Markov chain being in state a_i at time 0, and $p(0, 0, a_i) = 0$ given that initially the system is in a state $\notin S_N$. Finally, by the law of total probabilities, $p(n, k) = \sum_{i=1}^{|S|} p(n, k, a_i)$.

Now we are in a position to derive $E(T_N)$. Let r denote the time the system first enters a state $\in S_N$ and $T_N(t) = \min(t, r)$. Analytically,

$$\begin{aligned} E[T_N(t)] &= \int_0^t (1 - P_{rs}(s))\, ds \\ &= \int_0^t \sum_{n=0}^{\infty} p(n, n+1) \cdot \frac{e^{-\Lambda s}(\Lambda s)^n}{n!}\, ds \\ &= \frac{1}{\Lambda} \cdot \sum_{n=0}^{\infty} \Big(1 - \sum_{j=0}^{n} \frac{e^{-\Lambda t}(\Lambda t)^j}{j!} \Big) \qquad (2) \\ &\quad \cdot p(n, n+1) \qquad (3) \end{aligned}$$

As $t \to \infty$, $E[T_N(t)]$ becomes the mean time for the system to enter a state $\in S_N$, i.e.,

$$E(T_N) = \lim_{t \to \infty} E[T_N(t)] = \frac{1}{\Lambda} \cdot \sum_{n=0}^{\infty} p(n, n+1). \qquad (4)$$

C. Results

Since the time is normalized to the time it takes to execute the group splitting/merging algorithm (1 unit of time $\sim$ 10 milliseconds), $\frac{1}{E(T_N)}$ ($\frac{1}{E(T_M)}$) represents the fraction of time the group splitting (group merging) operation is invoked and performed. As shown in Fig. 4, $E(T_N)$ ranges from 40 to 100 as the connection setup rate λ varies from 0.01 to 1 connections per unit time (i.e., 1–100/second) and the connection duration $1/\mu$ varies from 10 to 500 units of time (i.e., 0.1–5 seconds). It shows that under different circumstances (varying K, M, N and λ, μ), the minimum value of $E(T_N)$ is 40. In other words, in the worst case, every 40 times the execution time of the group splitting/merging algorithm, one splitting/merging is needed, which implies that the time required for group management is at most 2.5% of the total system time. This is corroborated by the empirical study in which we found the system time used to split/merge groups is less than 1% of the system time for all experiments performed.

V. Related Work

The TCP extension for transaction (*T/TCP* [14], [15], [16]) was proposed to cache the *RTT*, the maximum segment size (MSS), the congestion window size, and the connection counter to expedite the catching up (of the sending rate) of new, repeated connections in the slow start phase. Touch [17] further proposed to share congestion-related information, such as the congestion window size and the estimate of round trip time, among TCP connections. They also differentiated information sharing among existing TCP connections from caching of temporal information from previous connections.

TCP-Int was proposed by Balakrishnan *et al.* [19] in which all concurrent TCP connections destined for the same destination use the same congestion window. The congestion window increases when an acknowledgment is received by any connection and is halved whenever packet loss is detected by any connection. *Ensemble-TCP* proposed by Eggert *et al.* [18] leveraged this information sharing concept, and in addition, used caching of temporal information to avoid the three-way handshake operation and the slow-start phase for repeated connections.

Both *TCP-Int* and *ensemble-TCP* require the use of schedulers to schedule packet transmission so that TCP connections can share in some fair fashion the "quota" dictated by the congestion window. Another drawback of the single congestion window

approach is that if one connection incurs packet loss, the congestion window is halved. This may be too conservative, as it is equivalent to reducing the congestion windows of *all* connections by half. In contrast, *COCOON* enables multiple, concurrent connections to have their individual congestion windows, but coordinates their congestion control activities. This eliminates the need for a packet scheduler. Moreover, when a single connection incurs packet loss, only those which may potentially be responsible for the congestion adjust their congestion windows. Finally, *COCOON* considers how to regulate, based on the control information carried in TCP traffic, non-responsive UDP connections in a TCP-friendly manner.

TCP Fast Start [20], [21] used the cached information, such as the TCP congestion window and *RTT*, to expedite the start up of new connections. More packets can be sent in the slow start phase, but are marked with a higher dropping preference. In the case of congestion, these packets will be dropped by intermediate *TCP-Fast-Start*-aware routers. Router support is required to achieve the best performance.

In TCP trunking [22], a TCP trunk is first established and all the connections destined for the same host are multiplexed in the trunk. The major drawbacks of TCP trunking are that a TCP trunk has to be established with the exchange of several, additional control messages and that packets have to be scheduled in order to share among connections the bandwidth of a trunk in some fair fashion.

The work that comes closest to ours is *Congestion Manager* (CM) [23]. As discussed in Section I, a CM sender ensembles concurrent unicast flows with the same destination host into a single flow. All the flows are subject to a single congestion window governed by a window-based AIMD congestion control algorithm. In particular, packet loss from any of the flows leads to multiplicative decrease of the congestion window. A CM sender also employs a rate-based traffic shaper and a scheduler to regulate packet transmission and to apportion available bandwidth among flows. In addition, CM uses a loss-resilient protocol to elicit feedback information periodically from receivers about losses and network status. The major difference between *CM* and *COCOON* has been highlighted in Section I and will not be repeated here.

VI. Performance Evaluation

We have implemented *COCOON* both in *ns-2* and in FreeBSD 2.2.8, and conducted performance studies to (i) validate the proposed design in terms of packet loss ratio, frequency of retransmission timeouts, attainable throughput, and response time, and (ii) compare *COCOON* against *TCP-Reno*, *TCP-Int*, *CM*, HTTP/1.0, and HTTP/1.1.

In the simulation study, we compare *COCOON* against *TCP-Reno* and *TCP-Int* with respect to packet loss ratio and response time. *TCP-Reno* is used as the base scheme in which no ECM is employed. The reason for selecting *TCP-Int* for comparison is because in the *ns-2* distribution, *TCP Fast Start* has been included in the *TCP-Int* implementation as part of the mechanism to expedite the start up of connections. The combination of *TCP-Int* and *TCP Fast Start* contains essentially all the features of *Ensemble-TCP*.

In the empirical study, we compare *COCOON* against *CM*, HTTP/1.0, and HTTP/1.1 with respect to attainable throughput, frequency of retransmission timeouts, and response time.

A. Simulation Study

We consider network topologies with a single bottleneck link (e.g., Fig. 5 (a)), multiple bottleneck links (e.g., Fig. 5 (b)), and arbitrary network topologies. We use an assortment of traffic sources (mainly infinite-duration TCP, finite-duration TCP, and on-off UDP sources). Due to the space limitation, we only report on a small set of simulations which we believe is most representative. In spite of quite a number of system parameters (topology,

link capacity, buffer size, object size, distribution of destination hosts) and algorithm parameters (M, N, and c) involved, the results are found to be quite robust in the sense that the conclusion drawn from the performance curves for a representative set of parameters is valid over a wide range of parameter values (unless otherwise stated).

In the single-bottleneck network topology shown in Fig. 5 (a), there are one sender (server) and n subnets (each of which contains a client), where n varies from 10 to 100. Totally n concurrent TCP connections are established for file transfer. The link bandwidth and delay between the sender and R_1 are 10Mbps and 2ms, respectively, while those between R_1 and R_2 are 1Mbps and 20ms, respectively. The link bandwidth and delay between R_2 and the first half of receivers are 10Mbps and 2ms, respectively, and those between R_2 and the other half of receivers are 10Mbps and (2+δ) ms, respectively, where δ is set to 0, 2, 4, 6, 8, 10 or 12 ms to assess the capability of *COCOON* group management. Each router employs a FCFS, drop-tail buffer management policy and is equipped with buffers of size 20 packets. The packet size is set to 500 bytes.

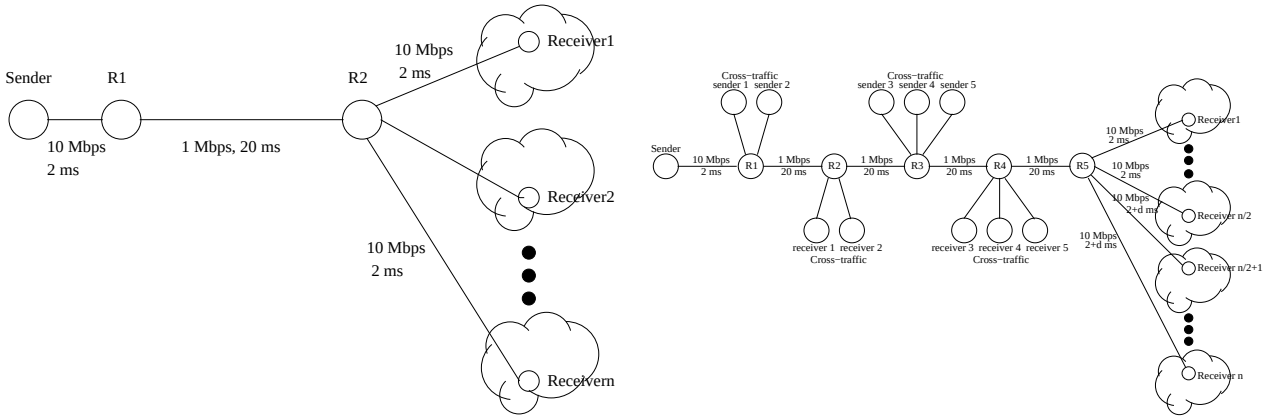
Two file sizes are used: 30 Kbytes (which is the average size of a webpage [13]) and 1 M bytes (which represents large file transfer). Transfer of 30-Kbyte files mimics the scenario in which multiple clients request web objects from the same server. Each connection starts at a random time that is uniformly distributed in $[0, 30]$ ms.

In the simulation runs for *TCP-Reno* and *COCOON*, one TCP connection is established between the sender and each receiver. On the other hand, to magnify the benefits of using *TCP-Int*, two TCP connections are established between the sender and receiver $2i$ in the simulation runs for *TCP-Int*, for $1 \leq i \leq \frac{n}{2}$, resulting in a total of $\frac{n}{2}$ flows.

In the multiple-bottleneck network topology shown in Fig. 5 (b), in addition to the n TCP connections mentioned above, there are other five infinite-duration TCP connections (labeled in the figure) traversing the bottleneck links between R_1 and R_2 and between R_3 and R_4, respectively. Finally, to demonstrate the capability of *COCOON* in regulating non-responsive UDP connections, in one of the experiments reported below, we replace 2 of the n TCP connections with non-responsive UDP connections with a total sending rate of 0.40 Mbps.

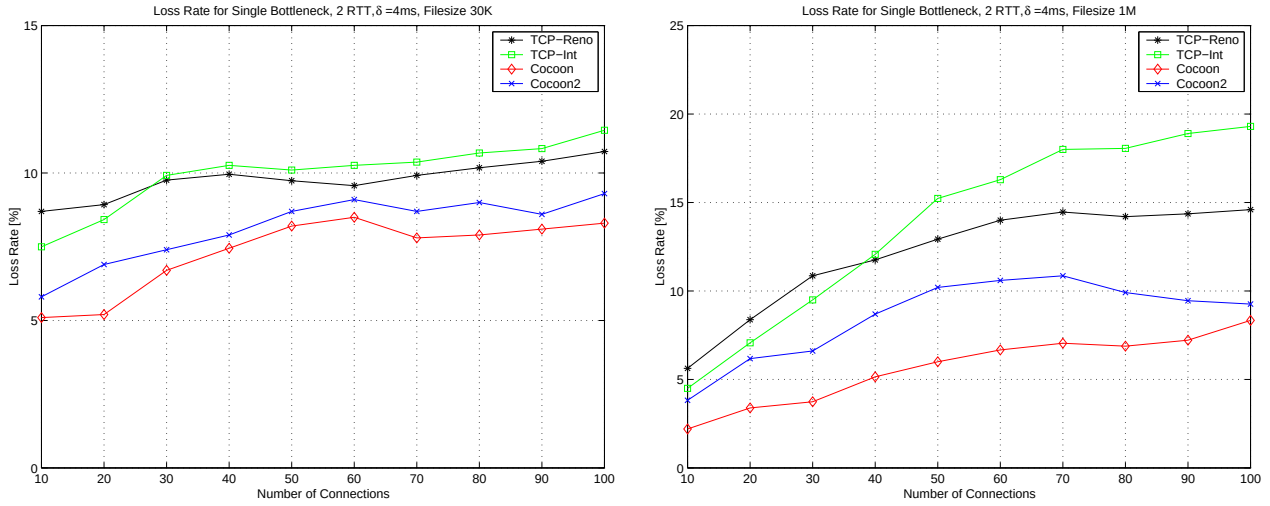
Performance in terms of packet loss ratio:: Figs. VI-A–7 give the performance of *TCP-Reno*, *TCP-Int* and *COCOON* with respect to packet loss ratio in both the single and multiple bottleneck topologies ($\delta = 4$ ms). *COCOON* groups the n TCP connections into two groups, with the first $n/2$ connections in the first group and the rest in the second group. The two curves labeled as *COCOON* and *COCOON2* in Figs. VI-A–7 represent the packet loss ratios experienced by the two groups. Under all cases, *COCOON* performs best, and the performance discrepancy becomes more significant as the number of concurrent connections increases or as the size of files transferred increases. The fact that *COCOON* outperforms *TCP-Int* suggest that the scope of coordination should not be restricted to connections destined for the same destination.

Performance in terms of response time:: Recall that a *COCOON* connection may reduce its congestion window and/or enter the congestion avoidance phase when some other peer connection incurs packet loss. That is, *COCOON* is proactive in congestion control and may not be as aggressive as TCP-Reno in utilizing available bandwidth. On the other hand, by coordinating congestion control among connections, *COCOON* incurs less packet losses (and hence less packet retransmissions and timeouts, as illustrated in Figs. VI-A–7). This, coupled with the fact that a new *COCOON* connection may begin with a larger congestion window, helps to sustain more throughput. To study which factors dominate, we measure the time it takes to transfer files of 30 K bytes and 1 M bytes using *TCP-Reno*,



(a) single bottleneck (b) multiple bottleneck

Fig. 5. The single and multiple bottleneck network topologies used in the simulation.

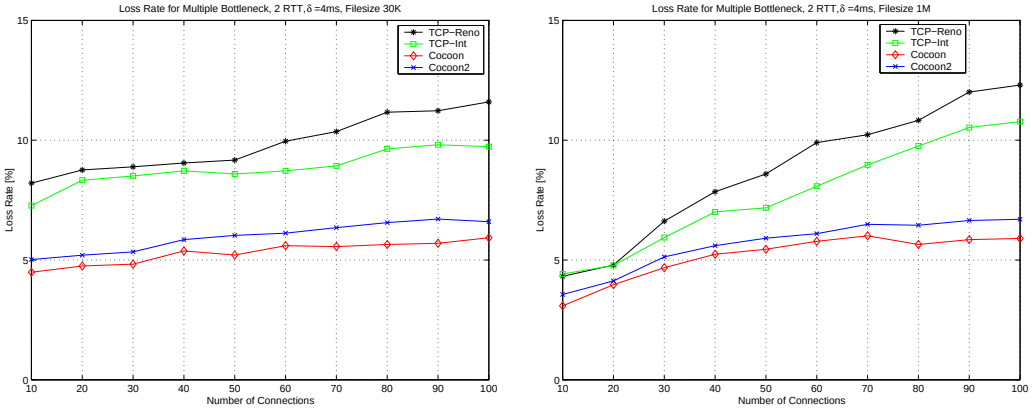


(a) 30 K bytes (b) 1 M bytes

Fig. 6. Packet loss ratio vs. number of concurrent connections in the single bottleneck topology.

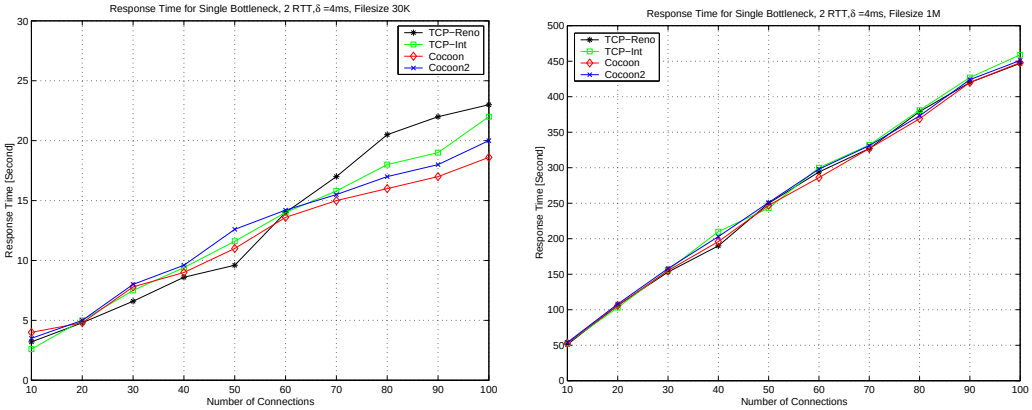
TCP-Int and *COCOON*. Figs. 8-9 give the results in both the single bottleneck and multiple bottleneck topologies, respectively. Again, the two curves labeled as *COCOON* and *COCOON2* represent the response time experienced by the two *COCOON* groups.

As shown in Fig. 8, the curves are very close to one another in the single bottleneck case. However, in the existence of interfering traffic (Fig. 9), *TCP-Reno* and *COCOON* outperform *TCP-Int*. This is because in *TCP-Int* all connections destined for the same destination host are treated as single flow that is subject to a single congestion window, so in the existence of interfering traffic, the total throughput sustained by *TCP-Int* flows is less than that in the other two schemes. In contrast, the n connections are still treated as n flows in *COCOON*, each of which is subject to its own congestion window (but shares the congestion information with the others in the same group). When a packet loss occurs, only connections (in the same group) that may potentially contribute to the congestion re-



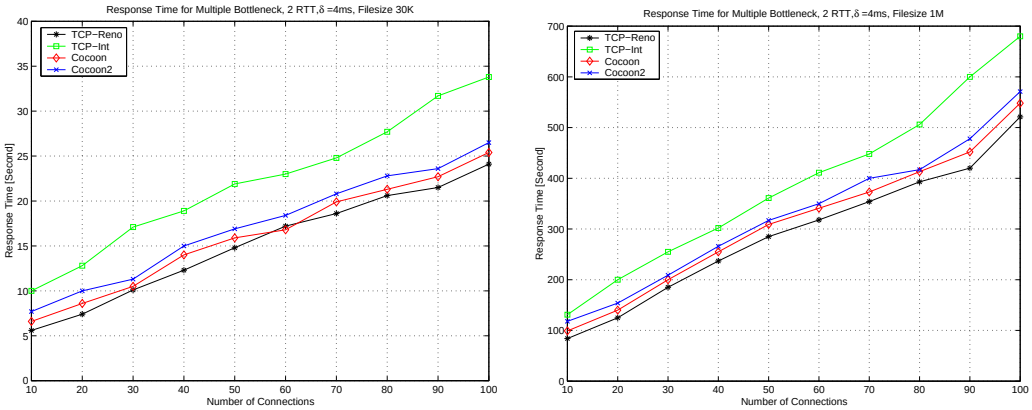
(a) 30 K bytes (b) 1 M bytes

Fig. 7. Packet loss ratio vs. number of concurrent connections in the multiple bottleneck topology.



(a) 30 K bytes (b) 1 M bytes

Fig. 8. Response time vs. # concurrent connections in the single bottleneck topology.



(a) 30 K bytes (b) 1 M bytes

Fig. 9. Response time vs. # concurrent connections in the multiple bottleneck topology.

duce their congestion windows. As a result, the total throughput sustained by *COCOON* flows is comparable to that by *TCP-Reno* flows. That is, from the perspective of users, a *COCOON* client do not perceive significant difference in the response time. However, from the perspective of network management, the bandwidth is better utilized under *COCOON* as a result of reduced packet loss ratio.

Performance in the co-existence of TCP-Reno, COCOON, *and* TCP-Int *servers::* To evaluate the performance when *TCP-Reno*, *COCOON*, and *TCP-Int* connections co-exist and compete against each other, we set $\delta = 0$ (i.e., all n TCP connections are subject to the same *RTT*s), and divide the n TCP connections into three groups, each of which running *TCP-Reno*, *COCOON*, and *TCP-Int*, respectively. Under *TCP-Int*, two TCP connections are established between the sender and receiver $6i$, for $1 \leq i \leq \frac{n}{6}$, resulting in a total of $\frac{n}{6}$ flows. Fig. 10 gives the packet loss ratio and response time incurred in sending files of 1 M bytes in the multiple bottleneck topology (in the presence of cross traffic). First, *COCOON* performs best in terms of packet loss ratio, because of its better coordination among concurrent connections, while *TCP-Reno* performs worst due to the blind competition among concurrent TCP connections. Second, *TCP-Reno* performs best in terms of response time, because each individual *TCP-Reno* connection probes and grabs the available bandwidth on its own. *TCP-Int*, on the other hand, gives the worst performance, because there are effectively $\frac{n}{6}$ *TCP-Int* flows competing against other connections (as well as against cross traffic) for the available bandwidth. The performance of *COCOON* lies in between, because similar to *TCP-Reno*, the $\frac{n}{3}$ connections are still treated as $\frac{n}{3}$ individual flows, but when a connection incurs packet loss, the other connections in the same group may have to backoff and hence do not grab the available bandwidth as aggressively as *TCP-Reno*.

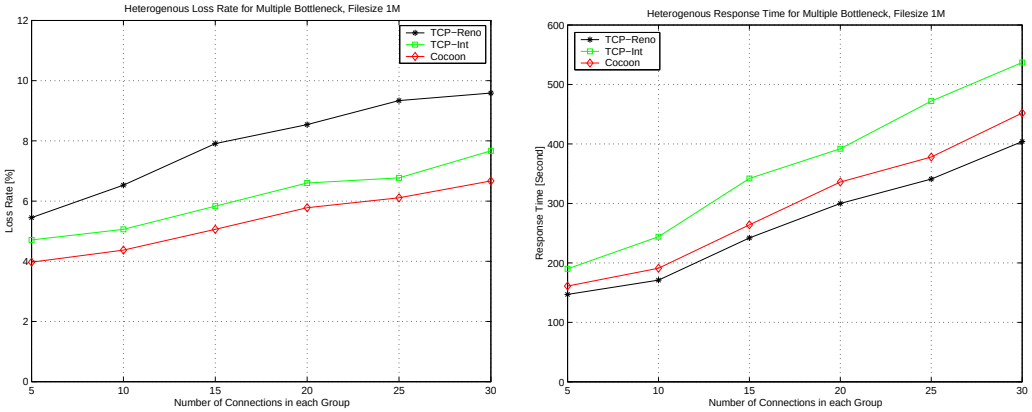
Fairness amongst Cocoon connections:: To study the impact of congestion coordination in *COCOON* on the fairness among connections in the same group, we repeat the experiments in Figs. VI-A–9 with $\delta = 0$, and calculate the fairness index defined in [24]

$$F(x) = \frac{(\sum_{i=1}^{n} x_i)^2}{n(\sum_{i=1}^{n} x_i^2)}, \tag{5}$$

where x_i is the throughput attained by the ith connection in a group. Fig. 11 depicts the fairness index under *TCP-Reno*, *TCP-Int*, and *COCOON*, respectively. (As the curves representing results obtained under the single multiple bottleneck topologies are indistinguishable, only the one for the multiple bottleneck topology is shown.) The fairness index for *COCOON* is inferior to that for *TCP-Int*, but is superior to that for *TCP-Reno*. This difference results from the fact that *TCP-Int* uses a scheduler to fairly regulate packet transmission among connections. *COCOON*, on the other hand, coordinates congestion control among connections in a group by requiring connections with large congestion windows to back off more in the case of packet losses.

Performance in the existence of non-responsive UDP connections:: We run the same set of experiments as in Fig. 11, except that 2 of the n TCP connections are replaced with non-responsive UDP connections with a total sending rate of 0.40 Mbps. Fig. 12 gives the TCP performance of *TCP-Reno*, *TCP-Int* and *COCOON* with respect to packet loss ratio and response time incurred in transferring files of 1M bytes in the existence of UDP connections and other interfering connections in the multiple bottleneck topology.

COCOON achieves the best performance in both measures because it regulates UDP connections in a TCP-compliant manner and prevents them from depriving TCP connections of their fair bandwidth share. *TCP-Reno* incurs the worst packet loss rate, due to the lack of coordination among the n TCP connections. On the other hand, *TCP-Int* incurs



(a) Packet loss ratio (b) Response time

Fig. 10. Performance in the co-existence of *TCP-Reno*, *TCP-Int* and *COCOON* connections.

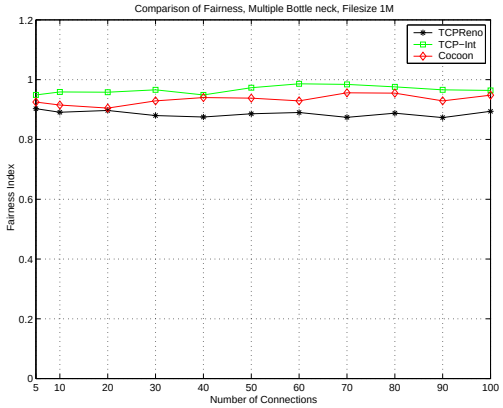
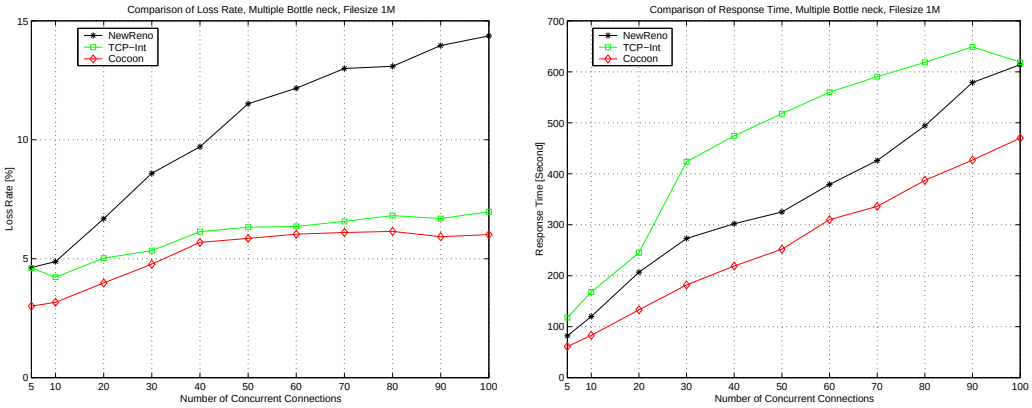


Fig. 11. The fairness index vs. the number of connections for transfer of files of size 1 M bytes.



(a) Packet loss rate (1M bytes) (b) Response time (1M bytes)

Fig. 12. Performance in the existence of non-responsive UDP connections.

Client host name	Client OS	Avg *RTT*
alpha.ece.ucsb.edu	SunOS 5.7	66.699 ms
rodan.ics.uci.edu	SunOS 5.7	67.101 ms
dsp7.eng.umd.edu	Digital UNIX	26.298 ms
hertz.ece.wisc.edu	SunOS 5.6	23.688 ms

TABLE II
THE HOST AND NETWORK CHARACTERISTICS FOR EXPERIMENTS RUN ON THE INTERNET.

the largest response time, because the number ($\frac{n}{2}$) of concurrent TCP connections in *TCP-Int* is less than that in the other schemes, and hence the total throughput attained in the presence of other UDP/interfering connections is also the least.

B. Empirical Study

We have implemented *COCOON* in FreeBSD 2.2.8. The interested reader is referred to [30] for a detailed account of the new data structures/functions created and the changes made in the existing kernel functions. We then carried out experiments over the Internet, with clients located at UCSB (alpha.ece.ucsb.edu), UCI (rodan.ics.uci.edu), UMD (dsp7.eng.umd.edu), and UW-Madison (hertz.ece.wisc.edu) and the server located at OSU (eepc121.eng.ohio-state.edu). The host and network characteristics are listed in Table II.

The experiments were carried out in 3 different time intervals (morning 8:00-11:00am, afternoon 1:00-4:00pm, and night 9:00-12:00pm) on a daily basis for a period of 2 weeks in September 2001. In each set of experiments, four methods are used:

HTTP 1.0 n HTTP requests of the form "GET file_name HTTP/1.0" are made from a client to the HTTP 1.0 web server, where n varies from 5 to 100. In this case, n parallel connections are established. No modification is made in the kernel at the server host.

HTTP 1.1 HTTP requests of the form "GET file_name_1 HTTP/1.1; ... ; GET file_name_n HTTP/1.1" are made from a client. In this case, one persistent connection is established. No modification is made in the kernel at the server host.

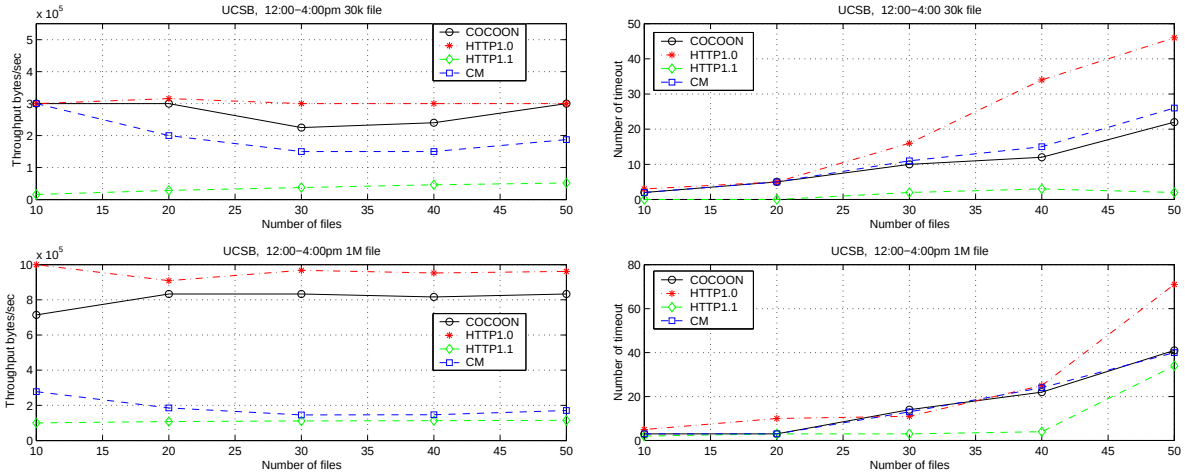
CM n HTTP 1.0 requests of the form "GET file_name HTTP/1.0" are made from a client to the web server. An implementation of *CM* alpha release is used at the server host.

COCOON n HTTP 1.0 requests are made from a client to the web server. An implementation of *COCOON* is used at the server host.

All the web servers run on an Intel Pentium II 400MHz machine with 128M byte memory. In each run, the throughput attainable by a client and the total number of retransmission timeouts were recorded. Five experiments were carried out for each combination of (# HTTP requests, method) in each time interval (i.e., each data point reported below is the average of over $5 \times 7 \times 2 = 70$ runs).

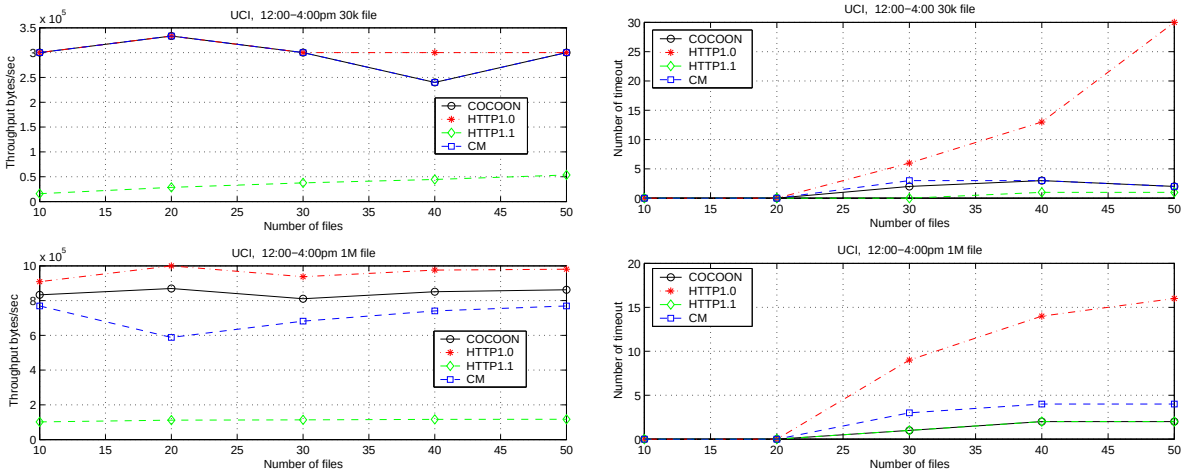
Figs. 13–16 give the empirical results for the experiments carried out in the afternoon time intervals, with the client being the host at UCSB, UCI, UMD, and UW-Madison, respectively. We show throughput and number of retransmission timeouts in those figures. The number of packet loss, which is indicated by duplicate ACK received by server, shows similar result as number of retransmission timeout, so it is not shown. Experiments carried out in the morning/night intervals give similar, but less pronounced results, and hence are not shown. We believe this is because the Internet carries more traffic in the afternoons and hence different congestion control methods can be better differentiated in this interval.

Several observations are in order: first, HTTP 1.1 and HTTP 1.0 achieve, respectively, the best and worst performance in terms of retransmission timeouts in all experiments. This is because with the persistent connection approach, HTTP 1.1 establishes one connection to sequentially transfer the n



(a) Throughput (b) # Retransmission timeouts

Fig. 13. Empirical results for HTTP requests initiated at UCSB.

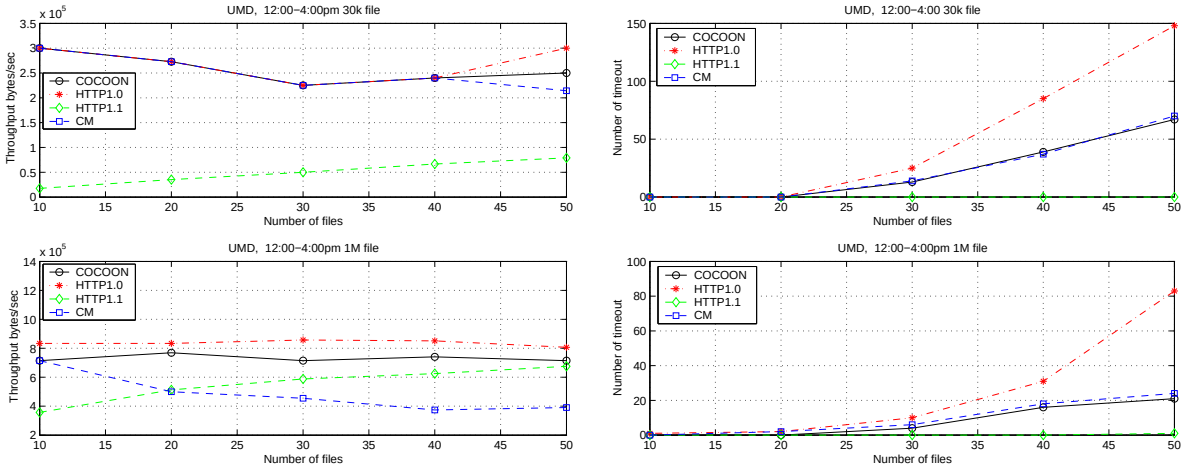


(a) Throughput (b) # Retransmission timeouts

Fig. 14. Empirical results for HTTP requests initiated at UCI.

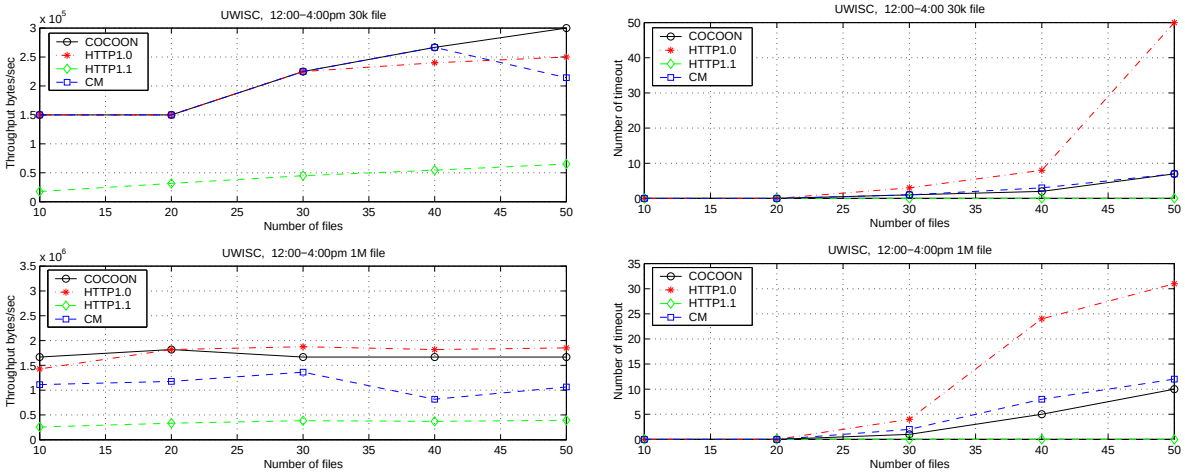
files. Timeouts occur only as a result of competition with other Internet traffic. HTTP 1.0, on the other hand, establishes n independent connections that compete blindly with one another as well as other Internet traffic, thus yielding the worst performance. Both *COCOON* and *CM* lie in between, with *COCOON* performing slightly better than *CM*.

Second, in the case of transferring large files, the performance in terms of attainable throughput (from best to worst) is HTTP 1.0, *COCOON*, *CM*, and HTTP 1.1. This is attributed to (i) both *CM* and HTTP 1.1 establish only one connection which has to compete with other Internet traffic. *COCOON* and HTTP 1.0, on the other hand, establish n individual TCP connections and hence are capable of grabbing more bandwidth shares; (ii) *COCOON* coordinates congestion control among group members, and hence is not as aggressive as HTTP 1.0 in grabbing bandwidth (but at the same time incurs less retransmission timeouts); and (iii) in the case of



(a) Throughput (b) # Retransmission timeouts

Fig. 15. Empirical results for HTTP requests initiated at UMD.



(a) Throughput (b) # Retransmission timeouts

Fig. 16. Empirical results for HTTP requests initiated at Univ. of Wisconsin.

packet loss in a group, *CM* reduces the composite congestion window by half (which is equivalent to reducing the congestion windows of all individual connections by half), while *COCOON selectively* reduces the congestion windows of those which may potentially be responsible for the congestion.

Third, in the case of transferring small files, HTTP 1.0, *COCOON*, and *CM* give similar performance in terms of attainable throughput (and all of them outperform HTTP 1.1). The may be due to the fact that the time taken to transfer small files is relatively small as compared to the time taken to perform three-way handshaking operations. As a result, the difference in the attainable throughput (which is amortized by the connection setup time) becomes insignificant. Furthermore, during the file transfer period, not many packet losses or retransmissions occur, so the benefit of ECM via *COCOON* or *CM* cannot be magnified.

To demonstrate the grouping capability of *CO-*

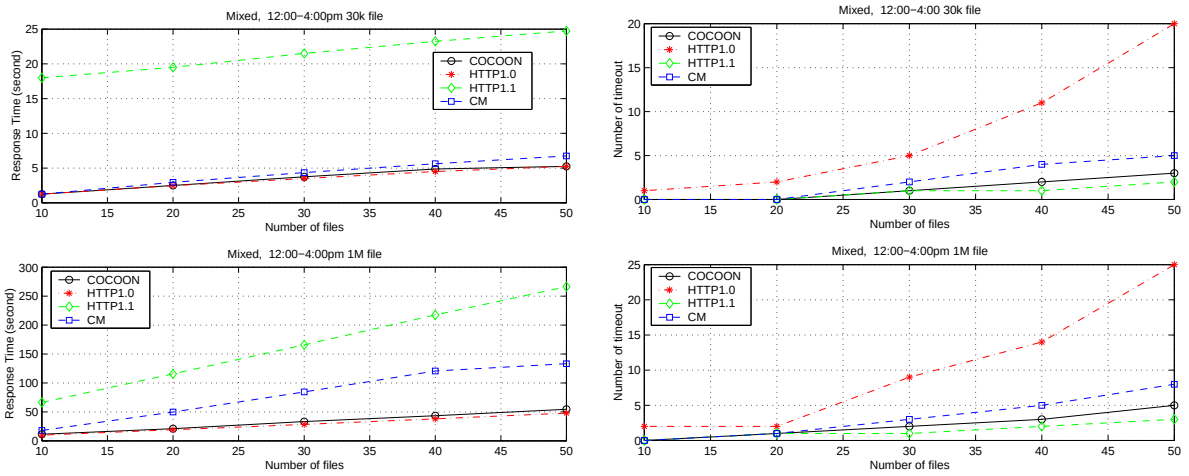
COON, we also carry out the following set of experiments (again in three time intervals on a daily basis over a period of two weeks): four clients from UCSB, UCI, UMD, and UW simultaneously make HTTP requests to the web server at OSU to transfer n files, where n varies from 10 to 50. Under this scenario, $4n$ (4) parallel connections are established under HTTP 1.0 (HTTP 1.1), $4n$ connections are established and grouped into 4 groups under *COCOON*, while $4n$ connections are ensembled into 4 flows under *CM*. Fig. 17 gives the empirical results for experiments carried out in the afternoon intervals. *COCOON* and HTTP 1.0 achieve comparable performance and outperform HTTP 1.1 and *CM* in terms of response time. Also, *COCOON* better coordinates congestion management among n connections in a group, as the number of retransmission timeouts is smaller than that incurred in HTTP 1.0 and *CM*.

VII. Conclusion

In this paper, we present an alternate endpoint congestion management scheme, called *coordinate congestion control* (*COCOON*). *COCOON* groups concurrent TCP and UDP connections destined for the same destination subnet/host, enables them to share congestion information (but not the congestion window), and coordinates congestion control activities among them. The size of a *COCOON* group can be dynamically adjusted, and the overhead of group management has been analytically shown to be reasonably small (no more than 2.5%) under a wide spectrum of subnet distribution and network loads. *COCOON* also allows a new connection to commerce with the (larger) congestion window of a reference connection in the same group. This expedites the start-up of the new connection but does not induce congestion. Finally, *COCOON* takes into account non-responsive UDP connections in a group and "bundles" them into a virtual connection which is subject to TCP-like congestion control.

COCOON requires modification only at the server hosts and can be readily deployed over the Internet. We have implemented *COCOON* in *ns-2* and in FreeBSD and performed simulation/empirical studies with respect to various network topologies and traffic loads. We observe that as compared to *TCP-Reno*, *TCP-Int*, *CM*, HTTP 1.0, and HTTP 1.1, *COCOON* indeed reduces packet loss ratio of concurrent connections, while sustaining throughput comparable to the best scheme. In particular, in the empirical study conducted over the Internet, we observe that there exists a trade-off between maximizing attainable bandwidth and minimizing packet loss. Approaches that treat all connections destined for the same destination as one flow and regulate them to a single congestion window (e.g., HTTP 1.1 and *CM*) avoids blind competition but attains less throughput in the existence of other Internet traffic. Approaches that do not coordinate congestion management (e.g., HTTP 1.0) attains the most throughput, but also suffers most from blind competition among individual connections (evidenced by the larger number of retransmission timeouts). The best approach is to treat connections as individual flows but coordinate congestion management activities among them.

We have also identified several avenues for further research. As discussed in Section II-D, if all the TCP connections in a group are inactive/closed, UDP connections in the same group may not respond to network dynamics, due to the lack of feedback from the network. We may artificially create a TCP connection that sends one-byte data packets periodically to probe the network status, and will investigate this approach with respect to both the control overhead thus incurred and the performance improvement thus achieved. We will also incorporate *COCOON* into HTTP servers as part of their congestion control mechanism to deal with congestion, the design and implementation details of which will be reported in a fore-coming manuscript.



(a) Avg. response time (b) # Retransmission timeouts

Fig. 17. Empirical results for HTTP requests simultaneously initiated at UCSB, UCI, UMD, and UM-Madison.

REFERENCES

[1] K. Thompson, G.J. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, November 1997.

[2] T. Berners-Lee *et al*. Hypertext Transfer Protocol – HTTP/1.0. RFC-1945, IETF, May 1996.

[3] R. Fielding *et al*. Hypertext Transfer Protocol – HTTP/1.1. RFC-2616, IETF, June 1999.

[4] T. Berners-Lee *et al*. The World Wide Web. *Communications of the ACM*, 37(8): 76–82, August 1994.

[5] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, Affordable, High-Performance Statistics Collection. http://www.nlanr.net/NA/Oc3mon/, August 1996.

[6] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW Client-based Traces. Technical Report, Boston University, CS Department, Boston, MA, April 1995.

[7] M. Arlitt, T. Jin Workload Characterization of the 1998 World Cup Web Site HP Technical Report, http://www.hpl.hp.com/techreports/1999/HPL-1999-35R1.html, September 1999

[8] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-Based Analysis of Web Object Sharing and Caching. *Proc. USITS'99*, October 1999.

[9] M. Allman. A Web Server's View of the Transport Layer. ACM Computer Communication Review, 30(5), October 2000.

[10] W. Stevens. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley, 1994

[11] M. Allman *et al*. TCP Congestion Control. RFC-2581, IETF, April 1999.

[12] UCB/LBNL/VINT Network Simulator - ns (version 2) http://www-mash.cs.berkeley.edu/ns/.

[13] B.A. Mah. An Empirical Model of HTTP Network Traffic. *Proc. INFOCOM'97*, April 1997.

[14] R. Braden. Extending TCP for Transaction - Concepts. RFC-1379, IETF, November 1992.

[15] R. Braden. T/TCP - TCP Extensions for Transactions. Functional Specification. RFC-1644, IETF, July 1994.

[16] W. Stevens. TCP/IP Illustrated, Volume 3: The Protocols. Addison-Wesley, 1998

[17] J. Touch. TCP Control Block Interdependence. RFC-2140, IETF, April 1997.

[18] L. Eggert, J. Heidemann, and J. Touch. Effects of Ensemble-TCP. *Computer Communication Review*, January 2000.

[19] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R.H. Katz. TCP Behavior of a Busy Web Server: Analysis and Improvements. *Proc. IEEE INFOCOM'98*, March 1998.

[20] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. *Proc. IEEE GlOBECOM '98*, Internet Mini-Conference, November 1998.

[21] V. N. Padmanabhan. Addressing the Challenges of Web Data Transport. PhD Dissertation, Univ. of California at Berkeley, September 1998.

[22] H.T. Kung and S.Y. Wang. TCP Trunking: Design, Implementation, and Performance. *Proc. IEEE ICNP'99*, October 1999.

[23] H. Balakrishnan, H.S. Rahul, and S. Seshan. An Inte-

grated Congestion Management Architecture for Internet Hosts. *Proc. ACM SIGCOMM'99*, August 1999.

[24] D. Chiu and R. Jain. Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks. *Journal of Computer Networks and ISDN*, Vol. 17, No. 1, June 1989.

[25] S. Savage, N. Cardwell, and T. Anderson. The Case for Informed Transport Protocols. *Proc. 7th Workshop on Hot Topics in Operating Systems*, March 1999.

[26] W. K. Grassman. Transient solutions in Markovian queuing systems. *Comput. and Ops. Res.*, vol. 4, pp. 47–53, 1977.

[27] D. Gross and D. R. Miller. The randomization technique as a modeling tool and solution procedure for transient Markov processes. *Operations Research*, vol. 32, no. 2, , March–April 1984.

[28] B. Melamed and M. Yadin. Randomization procedures in the computation of cumulative–time distributions over discrete state Markov processes. *Operations Research*, vol. 32, no. 4, pp. 926–944, July–August 1984.

[29] G. R. Wright and W. Stevens. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley, 1995

[30] Y. Gao. Implementation of *COCOON* in FreeBSD 2.2.8. Implementation notes, December 2001. Available at http://eewww.eng.ohio-state.edu/∼gaoy.

APPENDIX

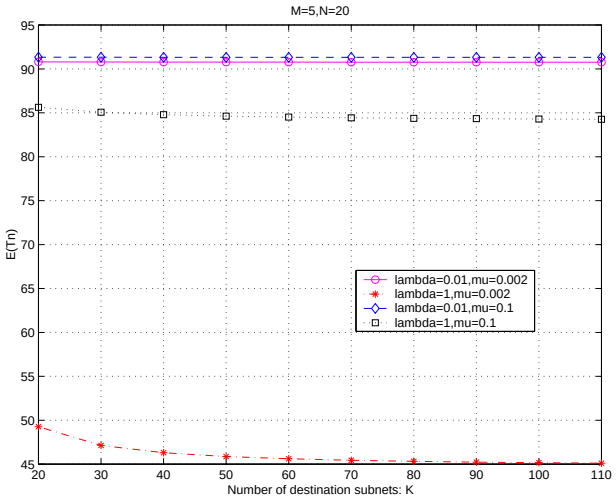
Randomization [26], [27], [28] is commonly used as a computational method to compute transient probabilities of Markov processes with finite state spaces. The main idea of this technique is to transform the Markov process into a Markov chain subordinated to a Poisson process as explained below.

Consider a Markov process $\{X(t), t \geq 0\}$ on a finite state space $S = \{0, 1, \ldots, \hat{N}\}$. The generator matrix, $Q = (q_{ij}, 0 \leq i, j \leq \hat{N})$, of the Markov process has the following property: $q_{ii} \stackrel{\triangle}{=} -q_i = -\sum_{j \neq i} q_{ij}$, where q_{ij} is the transition rate from state i to state j, and $q_i = \sum_{j \neq i} q_{ij}$ is the rate of leaving state i.

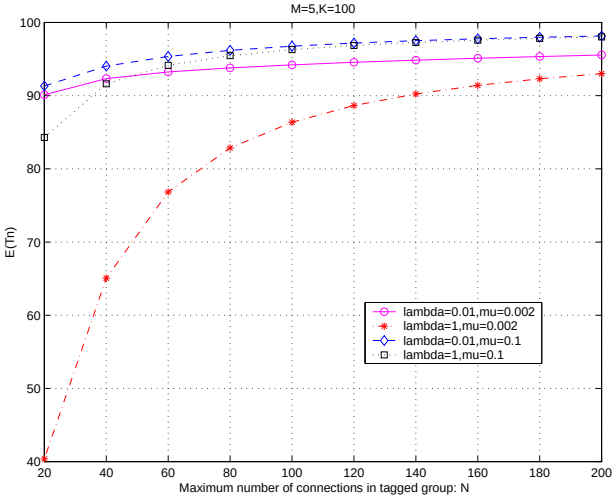
Let Λ be any value such that $\Lambda \geq q_i, \forall i$. At some instant, if the process is in state i, then it leaves state i at rate q_i, but this is equivalent to assuming that transitions occur at rate Λ, but only the fraction q_i/Λ of them are real (in the sense that the process really leaves state i and enters some other state and hence real transitions occur at rate q_i), and the remaining fraction $1 - \frac{q_i}{\Lambda}$ are fictitious transitions which keep the process in state i. That is, any Markov process can be thought of as being a process which spends an exponential amount of time with rate Λ in state i and then transitions to state j with a probability

$$P_{ij} = \begin{cases} 1 - \frac{q_i}{\Lambda}, & j = i, \\ \frac{q_{ij}}{\Lambda}, & j \neq i. \end{cases} \tag{6}$$

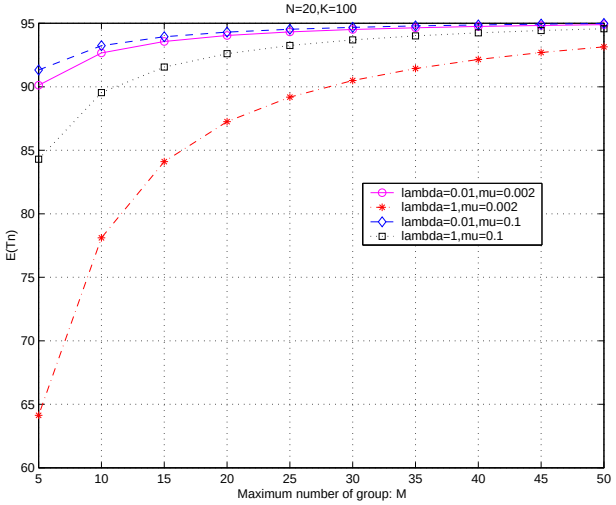
or in matrix form, $P = Q/\Lambda + I$. Consequently, there exists two component processes in a Markov process: a discrete-time Markov chain $\{Y_n, n = 0, 1, \ldots\}$ on S with transition matrix $P = Q/\Lambda + I$, and a Poisson process $\{N(t), t \geq 0\}$ with rate Λ. Moreover, $\{Y_n, n = 0, 1, ...\}$ and $\{N(t), t \geq 0\}$ are independent of each other and the process $\{Y_{N(t)}, t \geq 0\}$ has the same finite dimensional distribution as (and is thus probabilistically identical to) the original Markov process.



(a) $E(T_N)$ vs. K



(b) $E(T_N)$ vs. N



(c) $E(T_N)$ vs M

Fig. 4. $E(T_N)$ versus K, N, and M.