TCP-SMO: Extending TCP to Support Medium-Scale Multicast Applications

Sam Liang, David Cheriton

Abstract— **Scalable reliable multicast protocols have been a focus of recent research, tackling the problem of efficient reliable data delivery to an arbitrarily large number of receivers. Yet, the common applications of multicast, such as multi-point file delivery, or video streaming from a media server, typically only involve a moderate number of receivers, such as a thousand or fewer. Moreover, because of the limited deployment of these specialized multicast protocols, it is common for applications to instead use multiple TCP connections, one for each receiver, to implement multi-point delivery when feasible, causing a significant demand on the transmission server and the downstream links.**

In this paper, we describe a multicast extension to TCP, called the Single-source Multicast Optimization (SMO), that optimizes this case of multipoint delivery, providing the benefits of multicast together with the familiar features and API of TCP. Our results from experiments based on a Linux implementation and performed on a test-bed show that TCP-SMO requires just a modest extension to the TCP implementation and provides the scalable performance of multicast up to over a thousand receivers, thereby satisfying the common case requirements. In addition, used with TCP-RTM (Real-Time Mode), TCP-SMO also supports real-time multimedia multicast applications well.

Index Terms— **Multicast, TCP, Reliable Multicast, Transport Protocols, Scalable, Real-time Streaming.**

I. Introduction

TCP is the most extensively used transport-level protocol in the Internet. Years of enhancement and fine tuning has made it very efficient and robust. TCP is also congestion adaptive, which contributes significantly to the tremendous growth of the Internet. In addition, the fact that most firewalls exclude all traffic other than TCP further encourages the use of TCP. However, TCP was originally designed for one-to-one communication, not one-to-many.

To provide multipoint data delivery, it is currently common to use multiple unicast TCP connections from the source server to each receiver. For example, individual video/audio TCP streaming is usually done from a media server to a relatively large audience for a distance lecturing session. One problem with this approach is that it can cause high load on the source server. For example, in one incident, Yahoo's servers were overloaded trying to serve an audience of many thousands of viewers when they attempted to broadcast a Victoria's Secret fashion show over the Internet. As another problem, multiple unicast connections can cause unnecessary congestion over a common tail circuit when many receivers are located at a common campus.

Distributed Systems Group, Stanford University. Email: {sliang, cheriton}@dsg.stanford.edu.

To take advantage of IP multicast, the application developer is forced to abandon TCP and either use UDP over IP multicast or to use a specialized multicast transport protocol. With UDP, the developer often has to implement mechanisms to achieve in-order data delivery, error recovery, etc., recreating the mechanisms available in TCP. Even with real-time streaming, the compression representations, such as MPEG-2, make the stream loss and order sensitive, requiring the reliable transport properties.

By selecting a specialized reliable multicast protocols ([1], [2]), the developer confronts the problem that none of these protocols are widely deployed, limiting the deployment of the application or requiring that the application developer distribute the multicast transport protocol implementation along with the application. In some cases, these specialized protocols even require special support in intermediate routers, further complicating deployment, especially those designed for very large scale applications that may potentially have millions of receivers.

However, most point-to-multipoint applications in practice only involve fewer than a thousand receivers. For example, an online lecturing session or a corporate meeting supported by video/audio streaming typically involves hundreds of participants or receivers and rarely thousands. It is logistically difficult to get larger numbers of receivers all ready to receive at the same time except for the few events with truly mass appeal. Some researchers [5] have suggested that the sizes of multicast channels follow a Zipf [6] distribution, meaning that most multicast channels are small except for a few extremely popular ones, and such distribution may not change significantly in the future. Moreover, current practice is to distribute high-demand content hierarchically, from a central source to web cache servers that can then distribute to clients (say using an Akamai-like cache server [4]). Using this hierarchical structure, a million receivers can be handled while each web cache handles a thousand or fewer clients.

In this paper, we describe an extension to TCP to efficiently support multipoint data delivery that scales to approximately a thousand direct receivers. We call this approach *Single-source-Multicast-Optimization (SMO)* because it is simple optimization over multiple unicasts from a single source. Our experimental results show that TCP-SMO achieves good performance for one-to-many data communication, for both reliable and semi-reliable real-time applications, by leveraging IP mul-

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

ticast, while retaining the power of TCP, including its guarantee of in-order data delivery, fast retransmission, and congestion control. It scales to handle the common size of multipoint delivery and can be used hierarchically to handle arbitrarily large distributions. TCP-SMO requires a minimal addition to the application programming interface, making it straight-forward to adapt unicast applications.

The next section describes the overall design of TCP-SMO and our choices to some design issues such as acknowledgement (ACK) processing, round trip time estimation, retransmission policy, etc. Section III describes the implementation of TCP-SMO and its socket API. Section IV discusses some issues regarding channel membership management and session relay. Section V presents the experimental results for performance evaluations. Section VI discusses some related work. Finally, Section VII concludes the paper and discusses some future work.

II. Design of Multicast Extension to TCP

A. Overall Design

We focus on single source multicast applications and use the SSM channel model [7] [8]. A channel is defined by a (S, G) pair, where S is the address of the source server and G some designated multicast address.

In our multicast extension to TCP, the source server maintains a separate TCP unicast connection to each receiver, and it maintains a common multicast channel (S,G) that is associated with these unicast connections. The source server sends each packet to the multicast address G, and it is treated as if sent out on each of the separate TCP connections. For the source server, this setup appears as n TCP connections, one for each receiver, in addition to the multicast connection. Logically, the multicast is an optimized transmission of the data to all the receivers, with the individual TCP connections handling the recovery, etc. The ACKs from each receiver are received on each separate connection, causing unicast or multicast retransmission, as appropriate based on the number of receivers missing the data. More discussion on retransmission policies is presented in subsection II-E.

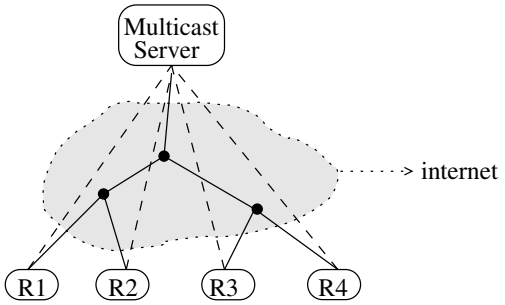

Fig. 1. Illustration of TCP-SMO. The solid lines represent the multicast distribution tree. The dashed lines represent the individual TCP connections between the server and the receivers. The black dots represent multicast routers.

At each receiver, this setup appears as a single TCP connection with the source. A multicast TCP packet received addressed with (S,G) is mapped to the associated TCP connection so that a packet is processed as though received addressed to this connection (after verifying the checksum and re-mapping the destination address).

Similar to the current common service model, such as that of an HTTP server or a FTP server, we adopt a *receiver-initiated connection model*. At the source, a server daemon waits for a connection from a subscriber [1]. For reliable data transfer, the server may choose to wait until a certain number of connections have been established before it starts sending data. Alternatively, especially for real-time applications, the server may continuously multicast data; after a new subscriber joins the channel and establishes connection to the server, it immediately starts receiving the data being multicast by the server. This model allows asynchronous join and leave of a session, which is particularly important for a multicast session of multimedia streams. Server-initiated connection model does not work for many situations, because it does not allow a user to join an on-going session asynchronously, such as an on-going lecture or a concert that is being multicast.

A basic requirement of using multicast is that all the receivers are ready and capable of receiving the same data at the same time. If any receiver is not able to keep up with the general multicast data delivery rate, either because it is too slow, or the path to it is too congested, it should either quit voluntarily or it should be removed by the source server from the multicast channel.

Going from unicast to multicast, a number of design issues arise. For example, how to efficiently manage the relationship between the multicast connection and the unicast connections for a channel, how the source server processes the acknowledgements from the multiple receivers and advances its send window, how it estimates the round-trip time for the entire audience as a whole, how the source server does retransmission when some receivers lose a packet, and so on. Next, we discuss these issues one by one.

B. Connection Management

For each multicast channel (S, G), a TCP-SMO source server creates a *master-socket* to manage the multicast connection. In addition, for each individual TCP connection to each subscriber to the channel, the source server creates a *child-socket*. These TCP connections are created to use a common sequence number for the data transmission to each subscriber.

Each *master-socket* contains a TCP Control Block (TCB) [9] that manages TCP state variables, such as sequence numbers, RTT, etc. We call this the *master-TCB* for the channel. Similarly, each *child-socket* contains a *child-TCB*.

The *master-TCB* keeps track of all the *child-TCBs* in the channel, as illustrated in Fig 2. When the *master-TCB* transmits

[1] In this paper we are only concerned with multicast applications, so we use the terms receiver, subscriber and client interchangeably, unless otherwise noted.

a packet to the multicast channel, it informs each *child-TCB* for each receiver of the transmission, so that they can update their control information, such as the last sequence number sent, etc. When a *child-TCB* receives an ACK, after it processes it itself, it forwards it to the corresponding *master-TCB* for further processing.

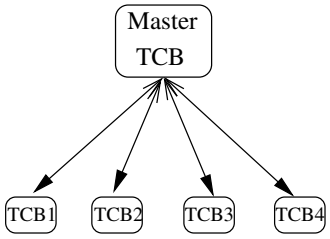

Fig. 2. *The relationship between the master-TCB and the child-TCBs for a multicast channel.*

C. Send Window Advancement

In a TCP sender's TCB, a pointer to the send sequence space $SND.UNA$ [9] keeps track of the next sequence number the sender expects to be acknowledged by the receiver. It also indicates the left edge of the send window. Data with smaller sequence number than $SND.UNA$ can be released. In the unicast case, $SND.UNA$ is advanced whenever an ACK acknowledging new data is received. However, in the multicast case, the problem of when to advance the master-TCB's $SND.UNA$ is more involved. There are three basic options to address this problem.

1) Wait for the slowest receiver. The *master-TCB* only advances its send window when a packet is acknowledged by all receivers within a reasonable amount of time. In other words, *master-TCB* only advances its send window after all its *child-TCBs* has advanced their send window. If a receiver does not acknowledge a packet for a prolonged period of time, it would be assumed too congested or dead, and could be removed from the channel.
2) Allow the application specify an ACK-count threshold in terms of a percentage of the session size that can advance the send window in the *master-TCB*. For example, the application can specify that if 90% of the receivers have acknowledged a packet, then that packet can be released and the send-window can be advanced.
3) Buffer-limited. If the *master-TCB's* send-buffer is full, remove the oldest packet and advance the window. One problem with this approach is how to determine the send-buffer size. One heuristic could be to use *expected-data-rate* $\times$ *max-RTT* as the buffer size.

Our design allows the application to specify one of these options, with the first one as the default.

Conceptually, these options specify a function F to compute the $SND.UNA$ value for the *Master-TCB* based on the input of the $SND.UNA$ values of the *child-TCBs*, as illustrated in Fig 3.

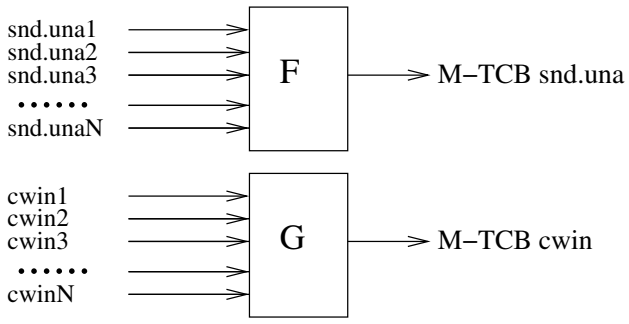

Fig. 3. *snd.una and cwin control functions. (M-TCB refers to the master-TCB.)*

For reliable data transfer, such as software or web cache distribution, the source server can be conservative and keep pace with the slowest receiver, in order to assure that all receivers successfully receive the exact copy of the data. Thus, these applications can usually use option one above.

For real-time applications, each receiver is expected and required to continue at the real-time rate, just delayed by the playback delay, so no receiver can fall excessively far behind. If one does, its connection may be dis-associated from the multicast connection, and it can choose to join a lower data rate channel. For example, a web proxy could deliver broadcast video to a large number of clients. For such real-time applications, usually option two or three can be chosen.

D. ACK Processing

One important issue that concerns reliable multicast researchers regards the processing of large number of ACK packets generated by the potentially large number of receivers. This has been termed as the *ACK-implosion* problem [10], [11].

However, risking being considered a heresy, we believe ACK implosion is not a significant overhead in all but very large collections of receivers. This is because there are fewer ACK packets than data packets (per connection) [2], and ACKs are small [3], easy to process and do not need to be buffered. Moreover, modern processors are capable of handling wire-speed packets on relatively high-speed network connections. Therefore, we believe for the medium-scale applications with a sub-thousand audience size, which TCP-SMO targets, ACK implosion is not a significant problem. Our experiment results presented in Section V show that a regular PC with a 930MHz CPU can handle 1200 receivers without overloading the CPU.

In addition, with the wide deployment of Gigabit (or even higher bandwidth) networks in the coming years, the ACKs from even a thousand receivers only take a small percentage of the available bandwidth. For example, if each receiver generates 25 ACKs/sec, corresponding to a data transmission rate of

[2] In a TCP session, there is typically one ACK for every two data packets.
[3] The size of a typical TCP ACK packet is 40 bytes.

50 packets/sec by the source server, the bandwidth used by the ACKs from 1000 receivers is only $25 \times 1000 \times 40 \times 8 = 8Mbps$, which is less than 1% of the bandwidth of a Gigabit network. Moreover, such traffic can be easily handled by the intermediate routers, which can process millions or even billions of packets per second.

Furthermore, the number of multicast servers to service a large audience will be relatively small (compared to the potential number of receivers), and such a server is reasonably expected to be located in a network with relatively high capacity and connected to relatively advanced routers. We do not expect such a multicast server to service a thousand receivers to be located behind a bandwidth-restricted tail circuit, such as a modem dial-up line. So such scenario is not considered in our design.

To further alleviate the ACK processing burden, we also propose an *Adaptive Acknowledgement* scheme to reduce the ACK frequency when no packet loss occurs. Currently, a TCP receiver typically generates one ACK packet for every two received packets. After the slow start phase, and when no packet loss occurs, a TCP receiver can send only one ACK for every four to eight received packets. As a result, the number of ACKs that the multicast server needs to process can be significantly reduced. To avoid the packet burstiness caused by reduced ACK frequency, TCP-pacing [12] can be used to smooth out the packet transmission.

E. RTT Estimation and Packet Retransmission

Each individual *child-TCB* maintains its own RTT and may timeout independently. The RTT for the multicast channel, maintained in the *master-TCB*, by default takes the maximum of the RTTs to all the receivers [4]. We think this default option can work reasonably well in most situations, especially in a LAN environment, where the typical RTT is below a few milliseconds. Alternatively the application can choose to specify a percentile number to use the maximum RTT of certain portion of the receivers, for example the maximum of the faster 90%.

Each *child-TCB* can perform fast-retransmit independently when certain number of duplicate ACKs are received, or it may retransmit a packet when its retransmission timer expires.

Whether to do unicast retransmission or multicast retransmission is determined by using a threshold number. This threshold number indicates the point where the server determines that unicast retransmissions would use more bandwidth than multicast retransmission, and when this threshold is exceeded, the server retransmits the missing packet by multicast. This threshold number (in terms of a portion of the session size) can be specified by the application, while the default value is set at 5%. The optimal value for this threshold number is a subject for further research.

F. Congestion and Flow Control

Each individual *child-TCB* maintains its own congestion window size ($cwin$) using the standard TCP congestion control algorithms [13], such as slow start, exponential back-off and congestion avoidance. The *master-TCB* derives the master $cwin$ from the $cwin$ values of the *child-TCBs* according to application configurable options. Such an option specifies a function G as illustrated in Fig 3.

By default, the master $cwin$ takes the minimum $cwin$ value of all the *child-TCBs*. This results in a multicast session that is *strictly fair* to all competing TCP unicast flows on all overlapping links. Similarly to the handling of RTT, if the $cwin$ of a connection is excessively small, that connection may be terminated and the associated receiver is sent a TCP Reset signal. Normally, if a receiver is not receiving data with sufficient quality, it should consider switching to a different channel, before the server dis-associates it from the channel.

Alternatively, we adopt the "essentially fair" concept proposed by Wang et al. [14] and provide an option to create an "essentially fair" multicast session. Basically, this scheme allows the bandwidth used by the multicast session to be different from that of a competing TCP flow, but bounded by certain parameters, as in the following equation:

$$a \times \lambda_{TCP} \leq \lambda_{TCP-SMO} \leq b \times \lambda_{TCP} \quad (1)$$

Where $\lambda_{TCP-SMO}$ is the bandwidth used by the TCP-SMO multicast session; λ_{TCB} is the bandwidth used by a competing TCP flow; N is the multicast session size; a and b are two bounding parameters, and $a \leq b < N$. a and b can be specified for TCP-SMO.

The rationale behind this scheme is that since a multicast session is servicing a large number of receivers, it may be acceptable for it to use slightly more bandwidth on an intermediate link.

In [14], λ_{TCP} is defined as the average throughput of a regular TCP connection from the source to a multicast receiver along the path with the smallest bandwidth. In TCP-SMO, because the source server keeps track of the congestion window size of each TCP connection to each receiver, it could use the smallest average congestion window size to indirectly infer λ_{TCP}.

Which option to choose depends on the specific application. If the application needs to be assured that the data is reliably transmitted to all receivers and is strictly fair to all competing TCP flows, it should transmit at the rate that the most congested receiver allows. However, for many real-time multimedia applications, they can choose to move on at a relatively regular pace according to the media being transmitted and the encoding schemes, and require receivers behind severely congested links to switch to lower data-rate channels [15] .

The receive window size for flow control is similarly handled as the congestion window size.

III. Implementation and Socket API

Our design principle is to make as little change as possible to either TCP semantics or its API, in order to achieve simpler implementation and straightforward deployment. In particular, we try to make it trivial for the application programmers to adapt

[4] If the RTT to a receiver is excessively longer than other receivers, this receiver may be dropped from the channel and its RTT ignored.

their programs to take advantage of IP multicast by using TCP-SMO.

Basically, the way applications construct the TCP-SMO server and receivers is exactly the same as regular TCP applications, such as an HTTP server and a web-browser. The only new steps are that the server needs to use a new socket option to specify the multicast channel information and control parameters, and that the receiver needs to subscribe to the channel before connecting to the server. If the application developer chooses to use the default TCP-SMO control parameters, he/she needs to add just one line each in the traditional TCP server and client programs, and can immediately take advantage of the benefits of multicast.

Next we delve into some technical details and describe the implementation of TCP-SMO and its socket API with an example.

At the source server, with address *S*, a server daemon creates a TCP socket for accepting TCP connections from the subscribers. We call this socket the *listening-socket*. Then it uses a new socket option to create a TCP-SMO socket associated with the *listening-socket*, which we call the *master-socket* [5], and to specify (in the same socket-option) a multicast group address G and a destination port number *MP* for this socket, which define the channel (S, G) with destination port *MP*. In addition, the application can also specify certain control parameters on the *master-socket* with this same socket option, such as the parameters that choose the retransmission policy, the RTT threshold to dis-associate a receiver, or the congestion window control policy, etc., as discussed in the previous section. The server daemon then waits for TCP connection requests from the receivers on the *listening-socket*, just as any regular TCP server program. After certain connections are set up, the server daemon can start multicasting data to the receivers by writing data to the *master-socket*.

At the receiver, the application creates a regular TCP socket, then subscribes the socket to the channel it is interested in, for example (S, G) using standard socket options, which are being used today by multicast applications based on UDP. Next, it just connects to the server as usual and starts receiving data from the socket that is multicast by the server. On the same receiver host, multiple sockets can subscribe to the same channel and receives multicast packets.

The following uses a simple example to further describe the API as well as the detailed protocol stack changes. For illustration, we use a sample topology with one source S and two receivers *R1* and *R2*, as in Fig 4.

First we define some abbreviations:

- *S*: the server's source address.
- *SP*: the server's source port.
- *G*: the channel's multicast address.
- *MP*: the channel's multicast port.[6]
- *RA1*: receiver R1's address.

[5] This is the same as the *master-socket* introduced in the previous section.

[6] For simplicity, we recommend, but not require, making *SP* the same as *MP*, so that when advertising the multicast channel, only *S*, *G*, and *MP* need to be advertised.

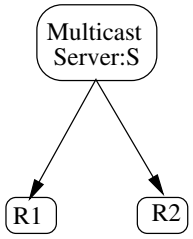

Fig. 4. *Sample topology to illustrate TCP-SMO's operation.*

- *RP1*: receiver R1's port.
- *RA2*: receiver R2's address.
- *RP2*: receiver R2's port.

A. At the Source Server

The source host follows the following steps to set up the server, take connections from the clients and transmit multicast data to the clients.

1) Use system call socket() to create a regular TCP socket, which is the *listening-socket*.
2) Call bind() to bind the socket to *[S, SP]*.
3) Use a new socket-option *TCP_SMO* on the *listening-socket* to create a new socket, called the *master-socket*, which contains a *master-TCB* that controls the multicast TCP operations. This socket-option also specifies the *master-socket* to use *G* and *MP* as its destination. Optionally, this socket-option may contain some control parameters to control the operation of the *master-TCB*.
4) Call listen() and accept() on the *listening-socket* to wait for connection requests.
5) When *S* receives a TCP SYN packet from R1, whose 4-tuple [7] is *[S, SP, RA1, RP1]*, *S* creates *TCB1* for *R1*, and replies with a SYN/ACK. *TCB1* contains a pointer to its parent, the *master-TCB*, and *TCB1* is added to a table in the *master-TCB*. This table is used to manage the relationship between the *master-TCB* and the *child-TCBs*, as illustrated in Fig 2.
6) When *S* receives a TCP SYN packet from R2, whose 4-tuple is *[S, SP, RA2, RP2]*, it creates *TCB2* for *R2*, and also replies with a SYN/ACK. The relationship between *TCB2* and *master-TCB* is similarly set up as in the previous step.
   The *master-TCB* owns a send-buffer. The children *TCB1* and *TCB2* don't have their own send-buffer, instead they share the *master-TCB's* send-buffer in a read-only manner. Also, *TCB1* and *TCB2* use the same sequence numbers that point to the send-buffer in *master-TCB*.
7) When *S* receives the ACKs from R1 and R2, the 3-way handshakes are completed, and the connections to R1 and

[7] A 4-tuple identifies a TCP connection. It contains the local source address, local source port, remote destination address and remote destination port. In this section, for clarity, we always list the server's address and port as the first two items in a 4-tuple.

R2 are successfully set up. The source server application can send data to the *master-socket*, which is multicast to the receivers. The data packet uses the multicast address *G* as its destination address, and *MP* as its destination port, but its protocol type is TCP instead of UDP.

8) ACKs from R1 are delivered to *TCB1*, and ACKs from R2 are delivered to *TCB2*. As described in sub-section II-D, these ACKs are also forwarded to the *master-TCB* to determine whether to advance the send-window in the *master-TCB*.
9) The connection tear-down is similar to regular TCP. The source server can close the socket connections to R1 or R2 at any time. The only extra step to do is to remove the leaving TCP connection (*TCB1* or *TCB2*) from the *master-TCB*. Also, when the server decides to disassociate a receiver for some reason, it sends TCP Reset packet to that receiver, which upon reception of the Reset packet should leave the channel.

B. At the Receivers

The receiver host follows the following steps to connect to the server and to receive multicast packets sent by the server.

1) Create a regular TCP socket.
2) Subscribe to the channel with *(S, G)* using standard socket options.
   Since TCP-SMO supports multiple sockets on the same receiver host to subscribe to the same channel and make connections to the server, the kernel needs to keep track of all the sockets that subscribe to *(S, G)* with a hash-table:*multicast-mapping-table* (*MMT*) indexed on *(S, G)*.
3) Send SYN with 4-tuple *[S, SP, RA1, RP1]* to the source server *S*.
4) Get SYN/ACK from *S*, reply with an ACK, set up TCB.
5) Receive a packet with 4-tuple *[S, SP, G, MP]*. Verify its checksum using the multicast address *G*. Look up *(S, G)* in the *MMT*, find all the sockets that subscribe to this channel, make a copy of the packet and deliver it to each of these sockets. The ACK sequence number in the multicast packet is ignored by the receiving socket.
   Each socket replies with an ACK independently.
   Note that such a receiver socket is able to receive both multicast packet and unicast packet that is retransmitted from the source server.
6) After the data transmission is finished, close the socket and un-subscribe from the channel.

In summary, the only API change is a new socket option at the server to specify the group address and the port number to specify the channel, and to specify some control parameters. Normally, the default parameters should be acceptable for most situations.

IV. Membership Management and Session Relay

A. Channel Membership Management

We use IGMP [16], [17] for membership management and require no new mechanism. A receiver joins the channel it wants to listen to before issuing a connect request to the source server. When the connection is closed, the receiver leaves the channel.

The server can send a TCP "Reset" message to a receiver to close the connection. When the receiver gets the "Reset" message, it should leave the channel.

As pointed out by Holbrook and Cheriton [7], the lack of convenient accounting capability is one of the major reasons that ISPs are not motivated to deploy IP multicast. Using TCP-SMO, the server has the knowledge of each receiver and can keep track of the number of receivers for accounting and billing purposes.

B. Session Relay

Session relay is used to support multi-source multicast applications, especially most-single-source applications, with the SSM model[8].

TCP-SMO supports session relay in a natural fashion. Each receiver can send any data back to the source server through the individual TCP connection. The source server can monitor the data and determine whether to multicast the data to all receivers.

This allows an end-receiver to actively participate in a multicast session. For example, in a live televised lecture session, a remote student can ask a question through the unicast TCP connection. When the teacher, who is at the source, receives the question, he/she can choose to repeat the question by broadcasting it to the entire class before he/she answers it.

V. Performance Evaluations

We created a prototype implementation [9] of TCP-SMO based on Linux kernel 2.2.12. Using this implementation, we conducted a series of experiments to study the performance of TCP-SMO regarding ACK processing, reliable data distribution and real-time multimedia data distribution, as well as its scalability.

A. ACK Processing

The first thing we would like to find out was whether TCP-SMO could handle the large number of ACKs generated by the potentially large number of receivers. On a LAN with 100Mbps bandwidth, we ran a TCP-SMO server program on a PC with a 930MHz CPU, and we created 1200 TCP-SMO receiver sockets evenly on five similar PCs[10]. These receiver sockets subscribed to a multicast channel rooted at the server machine. After the connection process phase, the server program started to multicast data packets to transmit a file of 36MB to the channel using TCP-SMO at a rate of about 100 packets/sec. These packets were received and acknowledged by the 1200 sockets, at the

[8]For more details, please see section 4 in [7].

[9]The congestion control part has not been implemented in the current prototype.

[10]Note that TCP-SMO supports multiple receiving sockets on the same host for a common channel. Also, such configuration does not affect the rate and the number that ACKs are generated, so it does not affect the result of the ACK processing at the server.

aggregate rate of about 50,000 to 60,000 ACKs/sec. These large number of ACKs were handled with ease by the server host; actually the CPU load caused by the TCP-SMO server program on the server host was lower than 0.1. After about 254 seconds, the transmission of the file was completed, and the 1200 receivers reported that they all received the complete file successfully.

When we raised the number of receivers to 1500, we observed that some TCP connections in the TCP-SMO session could not advance in a timely manner due to the loss of ACKs by the server host.

This result assured us that a regular PC [11] could easily handle the large number of ACKs from a thousand or fewer receivers. A server with more processing power, with a faster CPU or multi-processors for example, can certainly support a even larger audience.

B. Reliable Data Distribution

Applications such as software distribution and web cache distribution to cache servers need to do reliable data transmission to multiple destinations. Because data transmission throughput is a major metric for such applications, we performed a series of experiments to measure the throughput of TCP-SMO.

We used TCP-SMO to transfer a large file to N receivers simultaneously over a *100Mbps* LAN, where N ranges from 100 to 1000, Fig. 5 presents the measured throughput results of these transmissions. The figure shows that TCP-SMO can deliver a throughput over 10Mbps to 100 receivers by multicast. It also shows that even for a session size of 800 to 1000, TCP-SMO can still deliver a throughput over *1Mbps*. On the other hand, the theoretical throughput [12] by using multiple unicasts is substantially lower than TCP-SMO, and it drops further as the session size increases, while the throughput of TCP-SMO stays over *1Mbps*.

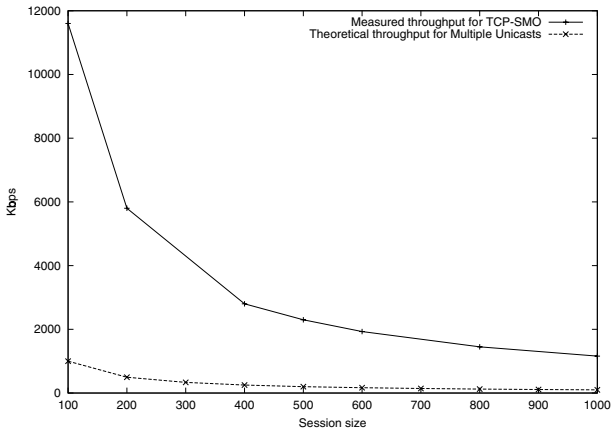

Fig. 5. *Comparison of the throughput between multiple unicasts and TCP-SMO.*

[11] A PC with 930MHz CPU, which we used in our experiments, is actually at relatively low end in today's market.

[12] The actually achievable throughput for multiple unicasts will be much lower in reality due to various overhead.

We observed that the maximum number of ACKs per second the server can handle was about 50,000 to 60,000, and this limited the throughput of TCP-SMO for large sessions. However, such throughput is still significantly higher than that of the multiple unicast approach.

All the above results were obtained by using the standard TCP ACK frequency, i.e. one ACK for every two data packets. When we reduced the ACK frequency to one ACK for every four data packets, we observed that the achievable throughput was basically doubled for each session size.

We also introduced artificial packet loss to some of the receivers to see if it would affect the TCP-SMO's throughput. In the experiment with 1000 receivers, when 400 of them suffered an independent loss rate of 2%, the multicast transmission using TCP-SMO still maintained a throughput at over *1Mbps*. So TCP-SMO can recover lost packets efficiently with fast-retransmits. The case that packet loss is bursty has not been studied systematically, and is left as part of our future work.

C. Real-time Multimedia Applications

Real-time multimedia applications have different characteristics from reliable data transmission applications. They are usually delay sensitive but may tolerate small packet loss rate. Traditionally, these applications use UDP as their transport protocol. However, [18] shows that a simple extension to TCP called TCP-RTM supports real-time applications well and provides good error-recovery. A major feature of TCP-RTM is that under lossy network conditions, it takes advantage of TCP's fast-retransmit feature to recover lost packets. In the case that a lost packet cannot be recovered by the time it is needed to be played back, TCP-RTM allows the receiver to skip over the lost packet and continue the playback using out-of-order packets. It provides the function to enable the receiver to trade full reliability for timely data delivery and low jitter.

To evaluate TCP-SMO's performance for real-time applications, we performed a series of experiments to use TCP-SMO along with TCP-RTM to multicast a simulated video data stream to 100 receivers simultaneously. This video data stream has a constant packet rate of 50 packets/sec, with the size of each packet being 1448 bytes, giving a data rate at about *0.58Mbps*.

For these experiments, we used a topology as depicted in Fig. 6. To emulate network latency, we used the Nistnet emulator [19] installed on a router between the server machine and the receiver machines. With the emulator, we conducted experiments that simulated receivers with network latency ranging from *20ms* to *100ms*. We also added packet loss simulation code in the Linux kernel's TCP receiving code and used that to simulate network packet drop rates ranging from 2% to 10% on each receiver [13]. Note that the packet loss on the receivers were not correlated, which actually caused higher load on the server because each lost packet needed to be retransmitted separately instead of retransmitted by multicast.

[13] We only simulated packet loss from the server to the receivers, but did not drop any ACK packet sent by the receivers to the server. More detailed results with two-way packet loss could be found in [18].

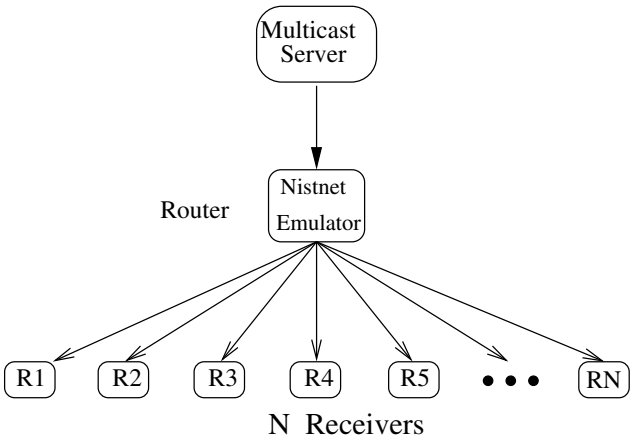

Fig. 6. *Experiment topology.*

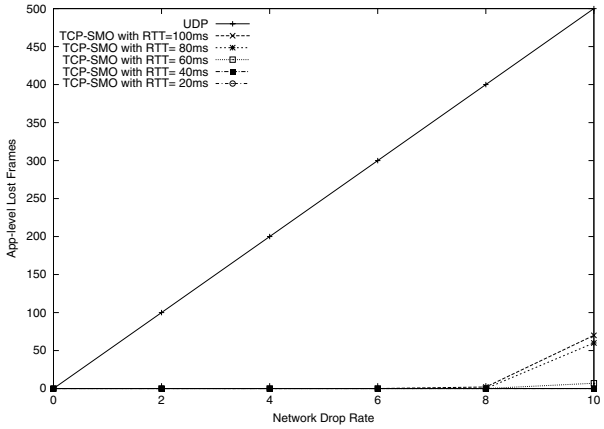

Fig. 7. *Application level-perceived number of lost frames vs network drop rates. 5000 packets (or frames) were transmitted for each experiment. The curves for various RTTs overlap with the X-axis for network drop rates less than* 8%

Fig. 7 shows the graph of the receivers' average perceived number of lost packets vs. the network drop rates. Out of the 5000 packets transmitted in each experiment, TCP-SMO along with TCP-RTM recovered almost all network dropped packets using TCP fast-retransmit, especially when network packet drop rate was less than 8%, performing substantially better than UDP.

These experiment results show that TCP-SMO along with TCP-RTM performs well for real-time multicast applications in terms of reducing the application-level perceived packet loss rate.

D. Scalability

We target TCP-SMO for medium-scale applications with fewer than a thousand receivers. But we believe larger scale applications can be supported by TCP-SMO as well by using a hierarchical structure, as illustrated in Fig. 8. At the first level, the root multicast server uses TCP-SMO to multicast data to the relay-servers (RS), which after receiving the data, use TCP-SMO to multicast it to the end receivers.

We conducted an experiment that used a two-level hierarchy. Three relay servers at the second level were connected to a root server using TCP-SMO. And these three relay servers used application level forwarding to forward data received from the root server to a total of 2300 receivers at the bottom of the hierarchy. Experiment results showed that a data rate of 1.1Mbps could be sustained from the root server to the end receivers. If more relay servers had been used at the second level, even higher data rate could be achieved.

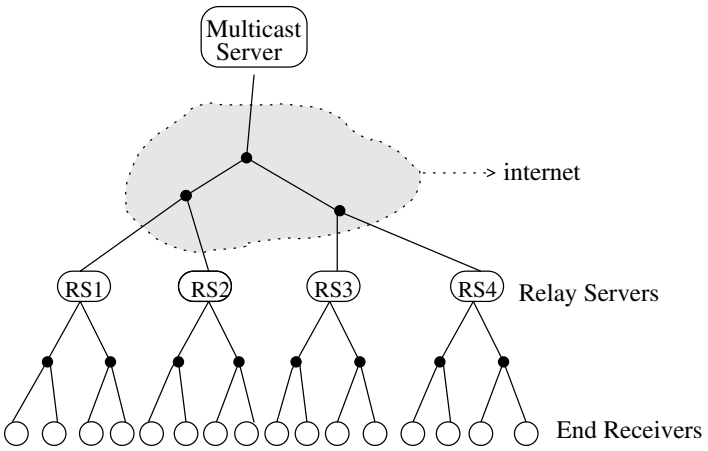

Fig. 8. *Illustration of a two-level hierarchical structure. Using second-level Relay Servers, substantially more end receivers can be supported.*

This experiment showed that TCP-SMO could scale up to applications with thousands of receivers. In theory, assuming one TCP-SMO server can support 1000 receivers at 1Mbps, a two-level hierarchy can support $1000 * 1000 = 1million$ receivers at 1Mbps.

These experiments in this section are far from comprehensive. More experiments need to be conducted to further evaluate TCP-SMO in more diverse situations. However, the experiments were representative and the results did suggest the potential and good performance of TCP-SMO for a wide range of multicast applications.

VI. Related Work

Back in 1988, Crowcroft et al. [10] described a theoretical multicast transport protocol that is "to provide a service equivalent to a sequence of reliable sequential unicasts between a client and a number of servers, whilst using the broadcast nature of some networks to reduce both the number of packets transmitted and the overall time needed to collect replies". TCP-SMO's design philosophy is in general consistent to theirs; in other words, TCP-SMO is designed as a straightforward multicast optimization over N unicasts in terms of data delivery.

Over the last decade, many reliable-multicast (RM) protocols ([1], [2]) have been designed and studied, such as SRM [20], XTP [21], LBRM [22], OTERS [23], MTP [24], PGM [25],

RMP [26], etc. This tremendous effort has obtained much better understanding of numerous issues and achieved significant progress on this topic. Some of these protocols perform well for certain applications. For example, SRM has been deployed on the M-Bone and supports well a large scale white-board application. On the other hand, despite such effort, the academia and the industry still lack a consensus on a standard deployable protocol that may meet the requirements of the common multicast applications.

Most of these reliable multicast protocols try to target very large scale multicast applications, having possibly millions of receivers. We think such lofty goal bring substantial complexity into their design. In contrast, we target some common medium-scale multicast applications, and rely on a hierarchical structure to service a larger scale audience when needed.

Also most of these protocols (with the exception of RMP and a few others) are mainly designed for fully reliable applications only. TCP-SMO, however, is designed to support both fully reliable multicast of bulk data and semi-reliable multicast of real-time multimedia data.

As far as we are aware, only a few of the RMs have real implementation. Many of the RMs only have a simulated implementation based on some network simulators such as ns [27], so their performance results on a real operating systems and in a real physical network are not available. In addition, we are not aware of any published results obtained from experiments conducted over a thousand receivers.

TCP-SMO is simple operationally. It does not involve the complicated setup of a DR (designated-receiver) tree and data recovery tree, which are required in a number of RMs. In addition, TCP-SMO is based on the mature protocol TCP. Therefore, we believe it is more robust than many other RMs.

Many RMs require retransmission from a DR or some other end-host other than the source server in the case of packet loss. This introduces serious security problems. In addition, for real-time streaming applications, since no receiver actually stores the data, so retransmission from any host other than the original server is difficult.

SCE (Single Connection Emulation) [28] is the closest to our approach. It uses a SCE sublayer between the unicast transport layer and the multicast network layer, which mimics the single destination network layer interface to the transport layer. It addresses a number of interesting design issues for a multicast transport service. However, unlike TCP-SMO, it uses a server-initiated connection model and does not allow asynchronous join, which is needed for real-time multimedia multicast. Also, it does not support TCP fast-retransmit and can only do multicast retransmission, which is inefficient in the case of independent packet loss. Finally, SCE does not allow a receiver to talk back to the source server through the SCE connection for the session relay purpose.

The Digital-Fountain approach [29] employs FEC and performs constant broadcasting of information. TCP-SMO can be used together with such approach. Because TCP-SMO reduces the application-level packet loss rate, it can reduce the redundancy required in the FEC coding and the associated encoding/decoding overhead.

Congestion control for multipoint data delivery is a complicated topic by itself [30], [31], [32], [33]. As pointed out by Wang et al. [14], even the meaning of *fairness* for a multicast session is still under debate. We believe TCP-SMO provides a simple and clean approach to address this problem as well as a convenient infrastructure to conduct further investigation.

A few commercial vendors provide commercial middleware to offer reliable multicast service. For example, GlobalCast [34] bases their products on RMP, SRM, etc. TIBCO [35], using a notion of "information bus", provides software that enables users to subscribe to groups representing classes of information in which they have interest. StarBurst [36] has patented MFTP [37] and is also deploying PGM [25].

VII. Conclusion and Future Work

TCP-SMO is a modest extension to TCP, the most extensively used unicast transport level protocol, providing multicast transport functionality in a simple and clean fashion. It makes available all the benefits of TCP, including in-order data delivery, fast-retransmit, congestion control and its extensively fine-tuned implementation, for multicast applications. It also preserves the TCP semantics and the existing TCP socket API, adding just one socket option. Such simplicity and full compatibility with TCP enables either easy adaptation of existing unicast applications or familiar development of new applications to take advantage of multicast transmission, achieving higher efficiency in the utilization of network and server resources.

While stretching the original domain targeted by the TCP designers, this TCP extension should be evaluated against the cost of a brand-new protocol to make a good engineering decision. It appears to provide the key benefits of multicast while largely retaining the TCP mechanisms, as we have described earlier.

The implementation of TCP-SMO is also simple. As a measure of its low implementation complexity, our Linux kernel implementation required fewer than a thousand lines of code change, about ten percent of the original TCP implementation.

Our experiment results show that TCP-SMO appears to offer good performance for target applications. First, it uses low network bandwidth. In the downstream direction, it takes advantage of IP multicast and substantially reduces the consumption of network resources, achieving far better results than N unicasts. In the upstream direction, the ACK overhead is not a serious problem in today's fast networks, especially for a medium scale audience with fewer than a thousand receivers, because the ACK packets take only a small percentage of the available network bandwidth, which will be even smaller when Gigabit networks are widely deployed. Second, TCP-SMO provides good performance for reliable applications with hundreds of receivers, such as software or web cache distribution, even under lossy network condition. Third, when used along TCP-RTM, TCP-SMO offers good performance for real-time multimedia applications in terms of achieving low application-level loss rates and low jitter, for receivers with various latency and network loss rates.

We also believe that TCP-SMO is scalable to support thousands of receivers with a hierarchical structure. As shown above, we performed experiments with good results that used application level forwarding at relay servers to support thousands of receivers in a two-level hierarchy. As part of our future work, we are considering to use TCP-level forwarding at the relay servers to reduce the forwarding delay involved in the application-level forwarding, which is particularly important for real-time multimedia applications. At a relay server, we plan to utilize TCP-forward ([38], [39]) with a pass-through TCP buffer. The pass-through buffer serves as both the receive buffer for receiving data from the higher level, and as the send buffer for sending data to the lower level. TCP-forwarding does packet forwarding at a level higher than IP-forwarding but below the application-level.

To further evaluate the performance of TCP-SMO, we would like to conduct more experiments in the public Internet, under more diverse conditions, such as with more burstily lossy links. We also plan to study the congestion control behavior of TCP-SMO in more depth. In addition, for incremental deployment, we are conceiving to create TCP-SMO middle-ware that can enable an application to receive multicast TCP packet without kernel modification. This way, only the multicast servers, which are much fewer, need kernel modification.

In summary, we believe that TCP-SMO leverages the power of TCP and meets the requirements of today's key multipoint data delivery applications. It offers numerous benefits with reasonable trade-offs, therefore it is an sensible approach that deserves further investigation and experimental deployment.

VIII. Acknowledgement

We would like to thank many other network researchers at Stanford University, and in particular Mark Gritter, Vince Laviano, Jonathan Stone, Dan Li and Xinhua Zhao, for their inspirational ideas and constructive comments. We would also like to thank the anonymous reviewers and Peter Danzig for their valuable feedback on our manuscript. This work was supported in part by DARPA under contract No. MDA972-99-C-0024.

References

[1] Walid Dabbous Christophe Diot and Jon Crowcroft, "Multipoint communication: A survey of protocols, functions and mechanisms," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 3, April 1997.

[2] Katia Obraczka, "Multicast transport protocols: A survey and taxonomy," *IEEE Communications Magazine*, Jan. 1998.

[3] Yatin Chawathe, "Scattercast–Taming IP Multicast: An Architecture for Internet Broadcast Distribution," *UC-Berkeley Ph.D dissertation*, 2000.

[4] "http://www.akamai.com," .

[5] Hugh W. Holbrook, "A Channel Model for Multicast ," *Ph.D dissertation, Computer Science Department, Stanford University*, 2001.

[6] G.K. Zipf, *Human Behavior and the Principle of Least-Effort*, Addison-Wesley, Cambridge, MA, 1949.

[7] Hugh W. Holbrook and David R. Cheriton, "IP multicast channels: EXPRESS support for large-scale single-source applications," in *SIGCOMM*, 1999, pp. 65–78.

[8] S. Bhattacharyya et al., "Internet Draft: An Overview of Source-Specific Multicast (SSM) Deployment," 2001.

[9] J. Postel, "RFC 793: Transmission control protocol," Sept. 1981, Status: STANDARD.

[10] J.Crowcroft and K. Paliwoda, "A multicast transport protocol," in *SIGCOMM*, 1988, pp. 247 – 256.

[11] George Varghese Christos Papadopoulos, Guru Parulkar, "An error control scheme for large-scale multicast applications," in *InfoCom*, 1998.

[12] Dennis Rockwell Joanna Kulik, Robert Coulter and Craig Partridge, "A Simulation Study of Paced TCP," *BBN Technical Memorandum 1218, Bolt, Beranek, and Newman*, August 1999.

[13] M. Allman, V. Paxson, W. Richard Stevens, "RFC 2581: TCP congestion control," April 1999.

[14] Huayan Amy Wang and Mischa Schwartz, "Achieving bounded fairness for multicast and TCP traffic in the internet," in *SIGCOMM*, 1998, pp. 81–92.

[15] Steven McCanne and Van Jacobson, "Receiver-driven layered multicast," in *SIGCOMM*, 1996.

[16] W.Fenner, "RFC2236: Internet Group Management Protocol, Version 2," Nov. 1997

[17] B. Cain, S. Deering, I. Kouvelas and A. Thyagarajan, "Internet Draft: Internet Group Management Protocol, Version 3," Feb. 1999.

[18] Sam Liang, David Cheriton, "TCP-RTM: Using TCP for Real Time Applications," in *http://dsg.stanford.edu/ sliang/rtm.pdf*, submitted for publication.

[19] Mark Carson, "Nistnet network emulator," in *http://snad.ncsl.nist.gov/itg/nistnet/*.

[20] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 784–803, 1997.

[21] J. W. Atwood, O. Catrina, J. Fenton, and W. Timothy Strayer, "Reliable multicasting in the xpress transport protocol," in *Proceedings of the 21st Local Computer Networks Conference*, October 1996.

[22] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton, "Log-based receiver-reliable multicast for distributed interactive simulation," in *SIGCOMM*, 1995, pp. 328–341.

[23] D. Li and D. Cheriton, "OTERS (On-Tree Efficient Recovery using Subcasting): A reliable multicast protocol," in *ICNP*, 1998, pp. 237–245.

[24] S. Armstrong, A. Freier and K. Marzullo, "RFC 1301: Multicast transport protocol," Feb. 1992.

[25] Tony Speakerman, et al., "Internet draft: PGM Reliable Transport Protocol, draft-speakerman-pgm-spec-06.txt," Feb. 2001.

[26] "http://research.ivv.nasa.gov/RMP," .

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

[28] R. Talpade and M. Ammar, "Single Connection Emulation (SCE): An Architecture for Providing a Reliable Multicast Transport Service," in *15th IEEE International Conference on Distributed Computing Systems*, 1995.

[29] M. Mitzenmacher A. Rege J.W. Byers, M. Luby, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," in *SIGCOMM*, 1998, pp. 56–67.

[30] S. Floyd, et al., "Requirements for congestion control for reliable multicast," in *Reliable Multicast Workship in Cannes*, September 1997.

[31] L. Rizzo L. Vicisano and J. Crowcroft, "Tcp-like congestion control for layered multicast data transfer," in *InfoCom*, 1998.

[32] Injong Rhee, Nallathambi Balaguru, George N. Rouskas, "MTCP: Scalable TCP-like Congestion Control for Reliable Multicast," in *InfoCom*, 1999.

[33] L. Rizzo, "pgmcc: A TCP-friendly Single-Rate Multicast Congestion Control Scheme," in *SIGCOMM*, 2000, pp. 17–28.

[34] "http://www.gcast.com, http://www.talarian.com," .

[35] "http://www.tibco.com," .

[36] "http://www.StarBurstCom.com," .

[37] K. Miller, K, Robertson, A. Tweedly, M. White, "Internet draft: StarBurst Multicast Transfer Protocol Specification, draft-miller-mftp-spec-03.txt," April 1999.

[38] D. Maltz and P. Bhagwat, "TCP splicing for application layer proxy performance," *IBM RC 21129*, 1998.

[39] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman and Larry L. Peterson, "Optimizing TCP forwarder performance," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 146–157, 2000.