RACCOOM: A Rate-Based Congestion Control Approach for Multicast[1]

Yuan Gao, Jennifer C. Hou and Sanjoy Paul

Abstract

As multicast applications have become widely deployed on the Internet, it is increasingly important to ensure these applications respond to network congestion in a TCP-friendly manner so as to coexist with TCP connections (which constitutes the majority of the Internet traffic). In this paper, we present a RAte-based Congestion COntrOl scheme for Multicast, called *RACCOOM*, for applications that deploy source-based multicast trees as the communication paradigm. In the absence of packet loss, a *RACCOOM* session keeps track of the congestion status of the on-tree path with the largest round trip time (called the *target* path), and adjusts its sending rate using a TCP Vegas [3]-like method. Upon detection of packet loss anywhere in the multicast tree, *RACCOOM* then responds by reducing its sending rate by half in a TCP-Reno manner. The ACK aggregation method used in *RACCOOM* prevents ACK implosion and yet provides the sender with a simple but comprehensive view of congestion conditions in the multicast tree. Finally *RACCOOM* is equipped with mechanisms to deal with changes of the target path due to traffic change and member join/leave.

To achieve TCP-friendliness, we have devised a simple method in *RACCOOM* to emulate how a TCP connection would behave under the same packet loss and delay characteristics. The results thus derived are used by *RACCOOM* to on-line adjust the parameters of its rate adjustment method. Alternatively, we can achieve (weighted) fairness (in terms of bandwidth sharing) among competing *RACCOOM* connections based on results obtained from feedback control theory. We validate the design, and demonstrate the features, of *RACCOOM* in ns-2. The encouraging simulation results, coupled with the fact that all the *RACCOOM* operations except acknowledgment aggregation (which requires modest router support) can be performed at end hosts, suggest that *RACCOOM* is a practical, and yet effective congestion control solution for multicast applications.

Index Terms — Rate-based congestion control, multicast, feedback control, TCP friendliness, weighted fairness, performance evaluation.

[1] Yuan Gao is with Department of Electrical Engineering, Ohio State University. E_mail:gaoy@ee.eng.ohio-state.edu.
Jennifer C. Hou is with Department of Computer Science, Uinversity of Illinois at Urbana-Champaign. E_mail:jhou@cs.uiuc.edu.
Sanjoy Paul is with Bell Laboratories Research. E_mail:sanjoy@lucent.com.

I. INTRODUCTION

We have observed an explosive growth in transporting data to multiple recipients on the Internet. As most Internet multicast applications do not support end-to-end congestion control, wide deployment of these applications have severe negative impact, ranging from starvation of self-controlled TCP connections to the potential for congestion collapse. The main intent of this paper is thus to design and evaluate a RAte based Congestion COntrOl scheme for Multicast, called *RACCOOM*.

There are several important issues we must consider in designing a multicast congestion control scheme. The first issue is support of the notion of TCP-friendliness and fairness among connections. Since multicast applications coexist with TCP-based applications , TCP-friendly congestion control prevents multicast connections from depriving self-controlled TCP connections of their fair share of the bottleneck bandwidth. TCP-friendliness means that if a connection shares a bottleneck link with a TCP connection, the connection should approximately receive, under the same delay and packet loss conditions, the same share of bandwidth as a TCP connection. Fairness, on the other hand, means that the bandwidth of a bottleneck link is fairly shared among all competing connections. Fairness considered in this paper is weighted fair-share, i.e. the bandwidth share allocated to a connection is proportional to the weight assigned to that connection. In some sense, weighted fair-share can be viewed as a means to express the relative importance of a connection.

The second challenge is scalability. Most congestion control schemes are either ACK-based, NAK-based, or a combination thereof. In the context of multicast, the ACKs/NAKs (which we collectively call ACKs) received by a sender from multiple receivers reflect diverse congestion conditions in various parts of the network, and have to be appropriately combined when making a single rate control decision. This leads to the problem of what information is carried in an ACK to best represent the congestion status and how ACKs are aggregated for scalability.

The third issue concerns judicious diagnosis of the congestion condition based on the ACKs received so as to isolate the effect of *independent* losses of the same packet in a multicast tree, A packet may be lost on one or more tree branches in a multicast tree, and as a result, several receivers may independently report loss of the same packet. If a sender reduces its sending rate for every such report, its sending rate will be severely throttled. This is termed as the *loss path multiplicity* problem in [2].

The next challenge is how to interpret and use round trip times (RTT) in congestion detection. Usually a prolonged RTT in unicast is a good indication of congestion at some intermediate router (which in turn serves as a preamble to packet loss)[2]. However, as there are multiple on-tree paths in a multicast tree, one for each receiver, it is neither practical nor scalable to keep track of RTTs along all on-tree paths. This leads to the problems of (i) along which on-tree path(s) the RTT(s) are kept track of and estimated, (ii) how the estimated RTT(s) are used in congestion detection, and (iii) what action to take if the RTT(s) on certain on-tree path(s) change because of dynamic traffic or membership change.

The last and not the least important issue is the mechanism used to adjust the sending rate. The two most widely used objectives in designing the rate adjustment mechanism are TCP-friendliness and fairness. No matter which objective is targeted, a good rate adjustment mechanism should be devised in an analytically provable manner.

In this paper, we propose, for source-base multicast trees, a new rate-based multicast congestion control approach, called *RACCOOM*, that addresses all the above issues. The rationale behind designing a rate-based approach is that as indicated in [7], the throughput attained by a window-based multicast congestion control scheme is constrained by the RTT of the farthest receiver, and is typically (much) less than the bandwidth available for all the receivers in the session. A rate based scheme, in contrast, is not subject to this constraint.

In *RACCOOM*, each ACK message contains, among other things, the sequence number, *LostSeq*, of the packet (if any) most recently detected to be lost, and the sequence number, *AckSeq*, of the packet most recently received. ACK messages are sent along the reverse path (on which data packets are sent) and are appropriately aggregated either by intermediate routers, or in the absence of router support, by non-leaf, designated receivers. If no packet loss is incurred, a *RACCOOM* session keeps track of the congestion status of the on-tree path with the largest RTT (called the *target* path), and adjusts its sending rate in a TCP Vegas-like manner [3]. Upon detection of packet loss anywhere in the multicast tree, *RACCOOM* then responds by reducing its sending rate in a TCP-Reno manner. By judiciously using parameters in the aggregated ACK and the local states kept at the

[2] Moreover, when the retransmission scheme is figured in, RTT is usually used to adjust the base timeout period.

sender, a *RACCOOM* session is able to identify independent losses of the same packet, to diagnose the reason of RTT changes (network congestion or dynamic traffic/membership changes), and to react correspondingly.

To achieve TCP-friendliness, we have devised a simple method in *RACCOOM* to emulate how a TCP connection would behave under the same packet loss and delay characteristics along the target path. The results thus derived are used by *RACCOOM* to on-line adjust the parameters of its rate adjustment method. As *RACCOOM* is a single-rate scheme, it cannot be TCP-friendly to every TCP connection that shares some link with it, but only connections that share the target path with *RACCOOM*. Alternatively, we can achieve weighted fairness (in terms of bandwidth sharing) among competing *RACCOOM* connections (on the target path) by tuning its parameters using results obtained from feedback control theory.

We have conducted *ns-2* simulation to test the TCP-friendliness, fairness, and scalability properties of *RACCOOM* and to compare *RACCOOM* against other existing approaches. We show *RACCOOM* can achieve TCP-friendliness, although it does not response to network congestion in exactly the same manner as TCP does. The encouraging simulation results, coupled with the fact that all the operations can be performed at end hosts,[3] suggest that *RACCOOM* is a practical yet effective congestion control solution for multicast applications.

The rest of the paper is organized as follows. In Section II, we present *RACCOOM* and its components. In Section III, we discuss how to fine tune the parameters of the rate adjustment method in *RACCOOM* to achieve either TCP-friendliness or (weighted) fairness among competing *RACCOOM* connections. In Section IV, we summarize and categorize existing work. In Section V, we present the simulation results and validate the proposed design. This paper concludes in Section VI with several avenues for future research.

II. *RACCOOM*

A. Acknowledgment aggregation for scalability

Every data packet (except retransmitted ones) sent by the source is associated with a unique sequence number and a time stamp (the time instant when the packet is sent). A receiver detects packet loss by observing a gap in the sequence number. When a receiver receives a packet, it sends an ACK message along the reverse path on which data packets are forwarded. An ACK message contains the following fields: (Fig. 1):

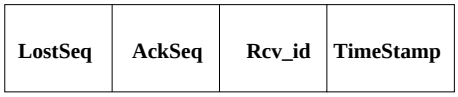


Fig. 1. Four fields in an ACK message.

1) *LostSeq*: The sequence number of the packet most recently detected to be lost (initially set to 0);
2) *AckSeq*: The sequence number of the packet received most recently (initially set to 0);
3) *Rcv_id*: The id of the receiver that reports *AckSeq*.
4) *TimeStamp*: the time at which the data packet to be acknowledged was sent. This information is copied from the data packet and used to estimate every packet's RTT.[4]

For example, if at the time a receiver receives a packet with sequence number 8, it has received packets with sequence numbers 1, 2, 4, 5, 7, then it sends an ACK message with *LostSeq=6*, *AckSeq=8*, its *Rcv_id*, and the time stamp when the packet with sequence number 8 was sent.

ACK messages are sent, and appropriately aggregated, along the reverse path to prevent ACK implosion at the sender. We design two versions of ACK aggregation mechanisms: one relies on router support, and the other on the help of *designated receivers*. Due to the space limitation, we only present the version that relies on router support.

1) Router-Assisted ACK *Aggregation:* In this version, every on-tree router aggregates the ACK messages received on its downstream interfaces, and sends upstream a single ACK message. The rules used to aggregate ACK messages are as follows. An on-tree router keeps, for each downstream interface, the latest ACK message received on that interface. (Initially the router keeps for each downstream interface a null message with $LostSeq = 0$, $AckSeq = 0$, $Rcv_id =$ *null*, and $TimeStamp = 0$.) It also keeps the latest ACK message sent upstream. Whenever a new ACK message arrives on one of its downstream interfaces, the router (i) replaces the old ACK message kept for that interface with the new one; (ii) composes an aggregated ACK by setting

[3] Provided that the underlying multicast routing protocol employs an *EXPRESS*-like method to count the number of downstream receivers.

[4] Note that no information of when the packets are sent is kept at the sender.

1) *LostSeq* to be the maximum value of *LostSeq*s received on all the downstream interfaces.
2) *AckSeq* to be the minimum value of *AckSeq*s received on all the downstream interfaces.
3) *Rcv_id* to be the id of the receiver that gives the minimum value of *AckSeq* in the composed acknowledgment.
4) *TimeStamp* to be the *TimeStamp* value in the ACK message with the minimum value of *AckSeq*.

If a new downstream interface is included (as a result of member join) when a multicast is in session, the router[5] initializes the message for that interface as *LostSeq* $= 0$, *AckSeq* $=$ the value of the *AckSeq* forwarded upstream, *Rcv_id* $=$ *null*, and *TimeStamp* $= 0$.

Rules for determining whether or not an ACK *message should be forwarded upstream::* When the value of *Rcv_id* in the newly composed ACK message differs from that in the last ACK message sent upstream, the ACK message is forwarded upstream. Otherwise, the ACK message is forwarded only when *LostSeq* or *AckSeq* is greater than the corresponding value in the last ACK message sent upstream. Note that the operations required in ACK aggregation are simple *min()*, *max()*, and *copy()* functions, and hence do not incur excessive processing delay (as compared to propagation delay). That is, the RTT estimated through aggregated ACKs represents the propagation and queuing delays.

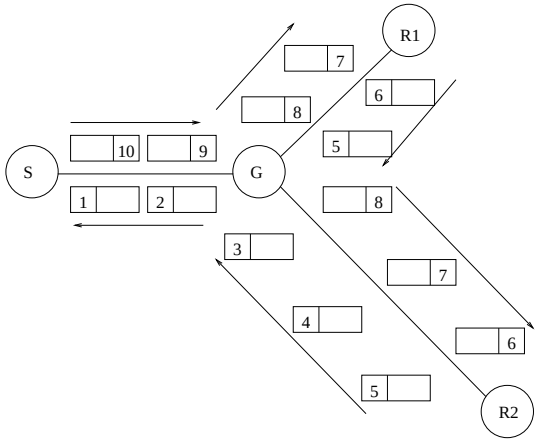


Fig. 2. An example of ACK aggregation. The boxes above and below a link indicate, respectively, the data packets and the ACK messages that are in transit on the link in the snapshot. The number inside a box indicates either the sequence number or the *AckSeq* value.

Example 1: Fig. 2 gives an example that shows how ACK messages are aggregated. *S* is the sender, *R1* and *R2* are the receivers, and *G* is an on-tree router. The round trip time from *S* to *R1* (*R2*) is approximately equal to the time to transport 8 packets (10 packets). In the snapshot of Fig. 2, *AckSeq* in the aggregated ACK message at router *G* is set to that in the ACK message forwarded by *R2* ($\min\{3, 5\} = 3$). *AckSeq* received by *S* is 1. Shown below are the *AckSeq* values received from *R1* and *R2* at router *G*, the *AckSeq* value forwarded upstream by router *G*, and the *AckSeq* value received at *S* at the kth packet time, $k \geq 0$.

	Time k	0	1	2	3	4	5	6
G	*AckSeq* received from *R1*							1
	AckSeq received from *R2*							
	AckSeq forwarded							0
S	*AckSeq* received							
	Time k	7	8	9	10	11	12	...
G	*AckSeq* received from *R1*	2	3	4	5	6	7	...
	AckSeq received from *R2*		1	2	3	4	5	...
	AckSeq forwarded	0	1	2	3	4	5	...
S	*AckSeq* received		0	0	1	2	3	...

As demonstrated in Example 1, because of the way in which ACK messages are aggregated, the *AckSeq*, *Rcv_id*, and *TimeStamp* values of an ACK message sent upstream (by an on-tree router) are received from the "most distant" receiver among all the downstream receivers. In other words, the *AckSeq*, *Rcv_id*, and *TimeStamp* values received by the sender are always those in the ACK message sent along the on-tree path with the largest RTT.

Several remarks are in order:

1) Storage overhead: a *RACCOOM* router does not keep all the ACK messages received downstream. Instead, for each downstream interface, it only keeps the *LostSeq*, *AckSeq*, and *Rcv_id* values of the latest ACK message received on that interface.
2) synchronization issue: a *RACCOOM* router does not have to wait for *ACK* messages to synchronously arrive on *all* of its downstream interfaces, in order to forward an *ACK* message upstream.
3) Responsiveness to member join/leave: *RACCOOM* does not keep track of downstream interfaces as receivers join/leave, but relies on the underlying multicast routing protocol to maintain such information. As a soft-state based multicast routing protocol may operate on a time scale that may be too coarse for congestion control (i.e., in the range of seconds), an on-tree router uses the following mechanism to detect ungracefully left receivers and to *timely* forward correct

[5] As will be discussed below, the router learns of the new downstream interface from the forward cache of the underlying multicast routing protocol.

AckSeq values: If an on-tree router finds that no *ACK* message arrives on one downstream interface, when each of its other downstream interfaces has received at least k *ACK* messages, it assumes that the receiver(s) on that interface have left and excludes that interface from ACK aggregation ($k = 3$ in the current implementation).

4) Message overhad: Recall that the values of *AckSeq* and *LostSeq* sent upstream are *always* non-decreasing. As an aggregated ACK message is sent upstream only when the values of *AckSeq*, *LostSeq*, or *Rcv_id* change, an on-tree router usually generates one aggregated ACK message per data packet. The only exception occurs when a member joins/leaves, and/or when a new packet loss is detected by one of its downstream receiver. In the worst case, one extra ACK message is generated and sent upstream for every instance of membership change and packet loss. The number of these instances is small as compared to the number of ACK messages.

2) Use of Aggregated ACK *Messages:* As indicated in [2], a path with a large RTT usually includes at least one link with long queues[6], and hence the congestion status along this path is usually more severe than other on-tree paths. Let this path be denoted as the *target* path. A *RACCOOM* session keeps track of the congestion status on the target path before packet loss takes place. On the other hand, when packet loss is detected anywhere in the multicast tree, it is reported in the *LostSeq* field, and by virtue of the way in which ACK messages are aggregated, will be delivered to the sender. This allows a *RACCOOM* sender to detect as early as possible any packet loss along *any* on-tree path. If the most congested link happens to lie on some path other than the target path, data packets that are routed along it may either experience increasingly long queuing delay, or eventually get dropped. In the former case, the target path will eventually change to the one that contains the bottleneck link. In the later case, as will be discussed in Section II-B, the *RACCOOM* sender will detect the packet loss (by perceiving *LostSeq* $\neq 0$) and halve its sending rate in a TCP-Reno manner. In either case, *RACCOOM* does respond to the congestion status of the bottleneck link.

The use of the *Rcv_id* field will be elaborated on when we discuss how *RACCOOM* handles dynamic traffic and/or membership changes. Finally, the *TimeStamp* field is used to estimate the RTT along the target path. When a sender receives an ACK message at time t, it calculates the current RTT estimate using the difference $t -$ *TimeStamp*, and will only keep the *minimum* value of RTT values estimated.

[6]Here we exclude the case of satellite links.

B. Rate adjustment

Fig. 3 outlines the procedure taken by a *RACCOOM* sender to adjust its sending rate before packet loss is detected. The sender keeps the minimum value, RTT_{min}, of RTTs that it has measured so far along the target path. In other words, RTT_{min} is the round trip *propagation* time along the target path under the condition that all the routers along the target path are queue free and all the packets on the path are in transit.

Let *SndMax* denote the maximum sequence number which the sender has sent. $N_{real} = SndMax - AckSeq$ is the total number of data packets outstanding in the network. Also, let R denote the sending rate and S the packet size, then $N_{exp} = \frac{R \times RTT_{min}}{S}$ is the (expected) number of outstanding packets under the queue free case.

If no queue is built up along the target path, the difference between N_{real} and N_{exp} should not be significant. On the other hand, when data or ACK messages are queued at congested routers along the target path, the difference between N_{real} and N_{exp} increases.

Based on the above observation, we use *diff* $= N_{real} - N_{exp}$ as an index of congestion status, under the case of no packet loss. Specifically, the sender adjusts its sending rate as follows:

$$R_a \leftarrow \begin{cases} R_a + \frac{k_2 \cdot |\textit{diff}| \cdot S}{RTT_{min}} & \textit{diff} < \alpha, \\ R_a - \frac{k_2 \cdot |\textit{diff}| \cdot S}{RTT_{min}} & \textit{diff} > \beta, \\ R_a, & \text{otherwise}, \end{cases} \quad (1)$$

and

$$R_{snd} \leftarrow R_a - k_1 \cdot \frac{\textit{diff} \times S}{RTT_{min}}, \quad (2)$$

where R_a is the *accumulated* rate, R_{snd} is the sending rate, and k_1, k_2, α, and β are all positive and tunable parameters. Intuitively Eqs. (1)–(2) intend to keep the number, N_{real}, of outstanding packets along the target path in the interval $[N_{exp} + \alpha, N_{exp} + \beta]$, or, equivalently, the number of packets that are *queued* at some intermediate routers (rather than in transit) along the target path in the interval $[\alpha, \beta]$. The

```
/* Upon receipt of an ACK message msg */
1.  RTT_est ← RTT_estimate();
2.  if (RTT_est < RTT_min)
    {
3.      RTT_min ← RTT_est;
4.      Rcv_id ← msg.Rcv_id;
    }
5.  N_real ← SndMax − AckSeq;
6.  N_exp ← (R_snd × RTT_min) / S;
7.  diff ← N_real − N_exp;
8.  if (diff < α)
9.      R_a ← R_a + (k_2 · |diff| · S) / RTT_min;
10. else if (diff > β)
11.     R_a ← R_a − (k_2 · |diff| · S) / RTT_min;
12. else R_a ← R_a;
13. R_snd ← R_a − k_1 · (diff × S) / RTT_min;
```

Fig. 3. Procedure taken to adjust the sending rate when packet loss is not detected.

rate adjustment method used here is similar to the window adjustment method used in TCP Vegas [3], except for the $k_1 \cdot \frac{diff \times S}{RTT_{min}}$ term in Eq. (2). The reason for including this term in R_{snd} is for system stability and will be elaborated on in Section III.

Note that although the sending rate is adjusted based on the amount of unacknowledged data reported by the receiver with the largest RTT, the throughput attainable by *RACCOOM* is not constrained by that receiver. This is because the rate adjustment scheme in *RACCOOM* uses, rather than the outstanding packet number (N_{real}), the *difference* between the outstanding packet number (N_{real}) and the expected outstanding packet number (N_{exp}) as an index of congestion.

Similar to TCP, the congestion avoidance phase commences when packet loss is detected (anywhere in the multicast tree) or upon acknowledgment timeout. Lines 2–7 in Fig. 4 give the procedure taken by a *RACCOOM* sender to adjust its sending rate in the congestion avoidance phase. In particular, to handle the *lost path multiplicity* problem [2] discussed in Section I, the sender keeps a parameter *LastCut* (initialized to zero) to keep track of the value of *SndMax* at the time when the sending rate is halved in response to packet loss or acknowledgment timeout. The sender reduces its sending rate only if *LostSeq* contained in the received acknowledgment message is non-zero and is greater than *LastCut*. Consequently, the sender reduces its sending rate at most once during one RTT interval. If the congestion persists beyond one RTT interval, the sending rate is reduced once every RTT, rather than once every packet loss. This mitigates the throughput degradation caused by the loss path multiplicity problem.

```
/* Upon session initialization */
1. LastCut ← 0;
2. Rcv_id ← null;
3. if ((LostSeq ≠ 0 && LostSeq > LastCut) || ACK timeout)
   {  /* congestion avoidance phase */
4.     R_a ← R_a/2;
5.     R_snd ← R_snd/2;
6.     LastCut ← SndMax;
   }
7. else
   {
       /* procedure taken to adjust the sending rate when packet
       loss is not detected. Figure 3 goes here */
   } /* else */
```

Fig. 4. Procedure taken by *RACCOOM* to adjust sending rate.

C. Handling of persistent congestion

If at the time when a *RACCOOM* session is set up, queues already develop at the routers on the target path (perhaps due to other traffic being carried over the network), the round trip times thus estimated will be considerably larger than the actual propagation delay of the path, leading to an overestimate of RTT_{min} (and hence N_{exp}). As a result, a *RACCOOM* session may set its sending rate to a value such that it believes its expected number of packets queued along the target path lies between α and β, when in fact it has sent more backlogged packets. This in turn worsens the congestion situation, and is termed as the *persistent congestion* problem. Similar situations have been reported to exist in TCP Vegas.

We deal with this problem by switching, upon packet loss, from the TCP Vegas-like rate adjustment method to the TCP Reno-like rate reduction method. When a *RACCOOM* sender overestimates RTT_{min} and sends more packets than it should, packet loss in the multicast tree occurs. The sender then reduces its sending rate by half, and if the congestion persists, it will further reduce its sending rate.

D. Capability to handle membership or network traffic change

Because members may dynamically join and leave a multicast group, the on-tree path with the largest RTT (i.e., the target path) may change as a result of member join/leave.

Similarly, because other traffic being carried over the network may change, the target path may also change as a result of traffic load change. In both cases, the value of RTT_{min} maintained by the sender should be updated so as for the sender to adequately adjust its sending rate. We discuss below how *RACCOOM* deals with each of the cases.

Handling of dynamic traffic changes:: Recall that the *Rcv_id* field in an ACK message is used to notify the sender of which receiver lies on the target path. When an on-tree router or a designated receiver aggregates all the ACK messages arrived on downstream interfaces, it sets *Rcv_id* to be the identifier of the receiver that gives the minimum value of *AckSeq* (and hence the most "distant" receiver). In the case of target path change, the *Rcv_id* field in the aggregated ACK will change as well.

To detect whether or not the target path has changed, a *RACCOOM* sender hence keeps a local state variable, *Target_rcv_id*, that keeps track of the receiver on the target path. If the *Rcv_id* contained in an ACK message differs from *Target_rcv_id*, it implies the target path has changed, and the sender should update both *Target_rcv_id* and RTT_{min} to reflect this fact. Specifically, the condition in which RTT_{min} is updated (line 2 in Fig. 3) is changed from "if ($RTT_{est} < RTT_{min}$)" to

$$\text{if } ((RTT_{est} < RTT_{min}) \,||\, (\textit{Target_rcv_id} \neq \textit{msg.Rcv_id})). \tag{3}$$

Handling of dynamic membership changes:: As discussed in Section II-A, when a member joins the multicast tree at an on-tree router, the underlying multicast routing protocol updates the associated (source, multicast group) entry of the forwarding cache. A *RACCOOM*-aware router can obtain this information from the forwarding cache and will be able to adequately aggregate ACK messages. In the case that the designated receiver-assisted approach is used, this information is passed to a designated receiver by its directly attached router. Because of the way in which ACK messages are aggregated, the *AckSeq*, *Rcv_id*, *TimeStamp* values received by the sender are those in the ACK message sent along the on-tree path with the largest RTT. If the on-tree path from the sender to the new member is not the one with the largest RTT, then this member join changes neither the *Rcv_id* field of the aggregated message nor RTT_{min}. On the other hand, if the on-tree path leading to the new member does become the one with the largest RTT, then the second test condition in Eq. (3) holds (because the *Rcv_id* field of the aggregated ACK contains the id of the new member). The sender will then update RTT_{min} and *Target_rcv_id* accordingly.

Similarly, the fact that a downstream member leaves (whether gracefully or not) is reflected in the associated entry of the forwarding cache kept by the underlying multicast routing protocol. (If a receiver does not leave gracefully, the downstream interface that leads to this receiver will be deleted upon state refresh timeout under the soft state approach most multicast routing protocols employ.) In the case that the leaving member is the most "distant" one (i.e., the leaving member is the leaf member to which the target path leads) the ACK message the sender receives will be routed along a new target path (and contains a new *Rcv_id*); the sender can update RTT_{min} and *Target_rcv_id* accordingly. In the case that a non-leaf member or a leaf member that does not lie on the target path leaves, the target path does not change and hence RTT_{min} will not be affected.

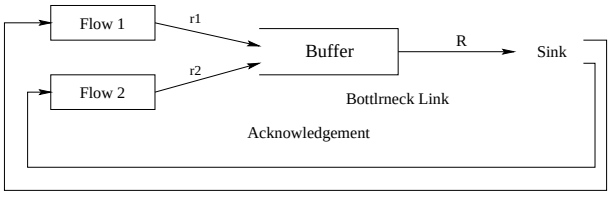
III. Analysis

In this section, we show how to set the parameters α and β in the rate adjustment method in order to achieve weighted fairness among competing *RACCOOM* sessions (Section III-A) or TCP-friendliness with co-existing TCP connections (Section III-B).

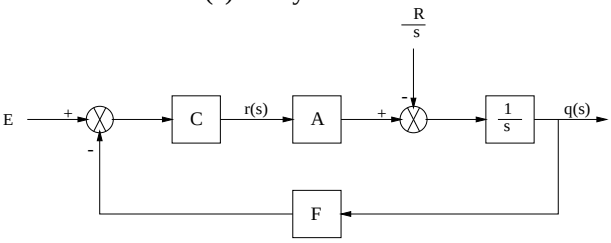
A. Setting parameters to achieve fairness

In this section, we use feedback control theory to analyze the behavior of *RACCOOM* before any packet loss occurs. Apart from leading to a better understanding of the control feedback capability of *RACCOOM*, the analysis also offers guidelines for setting the parameters α and β for achieving (weighted) fairness among *RACCOOM* sessions.

Fig. 5(a) depicts the system under consideration. We consider two flows, flow 1 and flow 2, that share the bottleneck link of capacity R on the target path. A flow may be a unicast or multicast session. In the latter case, the flow refers to the data stream that traverses the target path. For ease of analysis, we assume that the sources always have data to send and that packets are infinitely divisible, i.e., we use the fluid model. Let r_1 and r_2 denote, respectively, the rate at which flow 1 and flow 2 send their data packets. Both flows adjust, based on the acknowledgments sent by the receiver, their sending rates r_1 and r_2 using the *RACCOOM* rate adjustment mechanism. Let $q(t)$ denote the queue length at the bottleneck link at time t. Since the rate at which data arrives at the bottleneck link is $r_1(t) + r_2(t)$ and the link capacity is



(a) Analysis model



(b) Block diagram of the system under consideration

Fig. 5. Analysis model.

R, $q(t)$ can be calculated as

$$q(t) = \int_{-\infty}^{t} [r_1(s) + r_2(s) - R]^+ \, ds, \tag{4}$$

where $[a]^+ = \max(a, 0)$. When the sender of flow i receives an acknowledgment message from the receiver, it estimates the queue length, $q_i(t)$, of its data at the bottleneck. Let $q_i(t) \triangleq f_i \cdot q(t)$, where f_i is the fraction of flow-i data packets in the buffer queue. Because of the propagation and queuing delays incurred, the queue length estimated at time t is in fact $f_i \cdot q(t-d)$, where d is the round trip delay.

We analyze the rate adjustment method expressed in Eqs. (1)–(2) under the (reasonable) assumption that $\alpha_i \cong \beta_i$. The rate adjustment method can now be described as follows: it first calculates the difference between the expected queue length, E_i ($= N_{exp}$ in Section II-B), and the actual queue length, $q_i(t-d)$, at the bottleneck link and then adjusts its rate via

$$\begin{aligned} \mathit{diff} &= E_i - f_i \cdot q(t-d), \\ r_i(t) &= k_1 \cdot \mathit{diff} + \int_{-\infty}^{t} k_2 \cdot \mathit{diff} \; dt, \end{aligned}$$

where k_1 and k_2 are the rate gains used in the rate adjustment method.

After performing Laplace transform, we can represent the system in Fig. 5(b) with the following parameters:

1) $\mathbf{E} = \begin{bmatrix} \frac{E_1}{s} \\ \frac{E_2}{s} \end{bmatrix}$: the column vector of the expected queue lengths of flows at the bottleneck link, where E_i ($i = 1, 2$) is the queue length of flow i. Because *RACCOOM* operates by keeping the number of packets *queued* at the bottleneck link along the target path in the dead zone $[\alpha_i, \beta_i]$, without loss of generality, we set $E_i = \frac{\alpha_i + \beta_i}{2}$, $\alpha = E_i - \triangle$ and $\beta = E_i + \triangle$, respectively, where $\triangle$ is a small value.
2) $\mathbf{F} = \begin{bmatrix} f_1 e^{-d_1 s} \\ f_2 e^{-d_2 s} \end{bmatrix}$: the column vector of the feedback from the receiver, where f_i ($i = 1, 2$) is the fraction of data packets in the queue that are from flow i.
3) $\mathbf{C} = \begin{bmatrix} k_1 + \frac{k_2}{s} & 0 \\ 0 & k_1 + \frac{k_2}{s} \end{bmatrix}$: the rate controllers of flows.
4) $\mathbf{r(s)} = \begin{bmatrix} r_1(s) \\ r_2(s) \end{bmatrix}$: the column vector of the flow sending rates.
5) $\mathbf{A} = \begin{bmatrix} 1 & 1 \end{bmatrix}$: the adder that performs $r_1(s) + r_2(s)$.
6) R: the link capacity of the bottleneck link.

Now the rates of flows can be calculated as follow:

$$\mathbf{r(s)} = \mathbf{C}(\mathbf{E} - \frac{\mathbf{F}}{\mathbf{s}}(\mathbf{Ar(s)} - \frac{\mathbf{R}}{\mathbf{s}})) = \frac{\mathbf{1}}{\mathbf{s}}\mathbf{CF}[\frac{\mathbf{R}}{\mathbf{s}} - \mathbf{Ar(s)}] + \mathbf{CE}. \tag{5}$$

By plugging the expressions of $\mathbf{C}$, $\mathbf{F}$, $\mathbf{A}$, and $\mathbf{E}$ into Eq. (5) and solving for $\mathbf{r(s)}$, we have

$$r(s) = \frac{R \begin{bmatrix} (k_1 s + k_2) f_1 e^{-d_1 s} \\ (k_1 s + k_2) f_2 e^{-d_2 s} \end{bmatrix}}{s^3 + (k_1 s + k_2) f_1 s e^{-d_1 s} + (k_1 s + k_2) f_2 s e^{-d_2 s}} + \frac{R \begin{bmatrix} (k_1 s + k_2) E_1 \\ (k_1 s + k_2) E_2 \end{bmatrix}}{s^2 + (k_1 s + k_2) f_1 e^{-d_1 s} + (k_1 s + k_2) f_2 e^{-d_2 s}}.$$

Note that the term $k_1 \cdot \mathit{diff}$ is introduced to ensure the system will (eventually) reach equilibrium. (See [**?**] for the control theoretic base.) By properly selecting k_1, we can also shorten the time needed for the system to reach equilibrium.

By letting $t \to \infty$, we obtain the flow sending rates when the system reaches equilibrium:

$$\mathbf{r}(\infty) = \lim_{\mathbf{s} \to \mathbf{0}} \mathbf{sr(s)} = \left[\frac{\mathbf{R} \cdot \mathbf{f_1}}{\mathbf{f_1} + \mathbf{f_2}}, \frac{\mathbf{R} \cdot \mathbf{f_2}}{\mathbf{f_1} + \mathbf{f_2}}. \right]^{\mathbf{T}} \tag{6}$$

Eq. (6) indicates that the sending rate of a *RACCOOM* flow i is proportional to f_i. This is a nice feature which we can exploit to realize different levels of fairness. For example, to enforce (weighted) fairness between two flows that share the bottleneck link, i.e. $r_1(t) = \ell \cdot r_2(t)$, we set $f_1 = \ell \cdot f_2$. Since the control system operates by keeping the number of flow-i data packets queued at the bottleneck

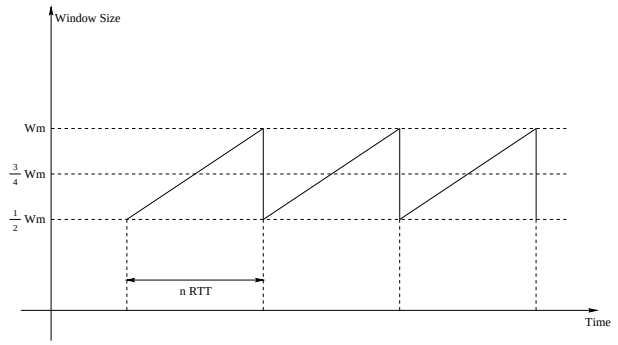


Fig. 6. Congestion window adjustment of TCP.

link as close to E_i as possible, we have $f_i = E_i/q(t)$. Hence, we can achieve (weighted) fairness by setting $E_1 = \ell E_2$. In the case that the traffic sources are of variable bit rate (VBR), $r_i(t)$ (and hence α_i and β_i) can be dynamically adjusted (on a coarse time scale) to match the data generation rate. The detailed algorithm is beyond the scope of this paper, and will be reported in a fore-coming manuscript.

B. Setting parameters to achieve TCP-friendliness

To achieve TCP-friendliness, a *RACCOOM* flow should adjust its desired queue length (i.e., the parameters α and β) to attain the throughput that a TCP connection would have received under the same packet loss and delay characteristics. When a *RACCOOM* flow detects packet loss, it adjusts its sending rate according to Eqs. (1)–(2). In addition, it also adjusts E (or equivalently α and β) as discussed below.

Fig. 6 depicts an instance of the window adjustment process of a TCP flow in the steady-state congestion avoidance phase (i.e., the slow start phase in the initial stage of a TCP flow and the effect of timeouts are not considered). In this phase, a TCP flow increases its congestion window linearly until it encounters packet loss, at which point it reduces its congestion window by half. The average TCP throughput is thus $\frac{3}{4}\frac{W_m}{RTT}$, where W_m is the maximum TCP congestion window size. In order to attain approximately the same average throughput as a TCP flow, a *RACCOOM* flow should adjust its sending rate so that it sends $\frac{3}{4} \cdot W_m$ packets per round trip time. That is,

$$N_{real} \cong \frac{3W_m}{4}. \tag{7}$$

Let the time interval between two consecutive packet losses be denoted as a period. From Fig. 6 we know that a period is of length n RTTs, where

$$n = \frac{W_m}{2}. \tag{8}$$

Combining Eq. (7) and Eq. (8), we have $W_m = N_{real} + \frac{n}{2}$. Let $E^{(k)}$ denote the desired queue length in the kth period and n_k the length (in units of RTTs) of the kth period. Then, when a *RACCOOM* flow reaches equilibrium and attains the same TCP throughput,

$$\begin{aligned} E^{(k)} &= N_{real} - N_{exp} = \frac{3W_m}{4} - N_{exp} \\ &= \frac{3}{4}(N_{real} + \frac{n_k}{2}) - N_{exp} \end{aligned} \tag{9}$$

Since in the previous period (i.e., the $(k-1)$th period), $N_{real} = N_{exp} + E^{(k-1)}$, we have

$$E^{(k)} = \frac{3E^{(k-1)}}{4} + \frac{3n_k}{8} - \frac{N_{exp}}{4}. \tag{10}$$

By keeping track of the time interval (in units of RTTs) between the $(k-1)$th packet loss and the kth packet loss (i.e., n_k), we can use Eq. (10) to adjust $E^{(k)}$'s (and hence the parameters α and β) upon every packet loss.

Note that fairness among competing *RACCOOM* sessions and TCP friendliness with co-existent TCP connections can be simultaneously achieved by having each *RACCOOM* session on-line adjust its parameters using Eq. (10). This is because under the above iterative approach, the on-line adjusted values of α and β will be approximately the same among all *RACCOOM* sessions. However, *weighted* fairness among *RACCOOM* sessions *with different weights* and TCP-friendliness cannot be simultaneously achieved.

IV. Related Work

Several TCP-friendly congestion control approaches have been proposed, among which the approaches proposed in [24], [22], [11], [21] are window-based and those in [15], [9], [18], [20], [25], [28] are rate-based.

Window based approaches:: All existing window-based approaches [24], [22], [11], [21] use the TCP congestion control and avoidance method to adjust their window sizes, whenever an ACK message is received or packet loss is detected. In what follows, we highlight their major differences.

SCE [24] is layered between TCP and IP. All the receivers acknowledge the data packets they have received.

When the number of ACK messages for a data packet exceeds a pre-determined threshold, a single ACK is generated and sent to the TCP layer. The major drawback of SCE is that it is not scalable, as a sender has to keep the states for all the receivers. IRMA [11] requires that multicast routers be modified so as to (i) keep track of numerous states for ACK aggregation, to (ii) estimate RTTs, and to (iii) support TCP semantics at end hosts. In MTCP [22], all receivers are organized into a multicast tree at the transport layer and data are reliably transmitted among them. The drawback is, however, that the resulting tree at the transport layer may not be optimal. PGMcc [21] emulates the evolution of the TCP congestion window when ACK or NAK from a selected representative receiver (called *Acker*) is received. The receivers report to the sender their estimated packet loss rates in their NAKs. The sender estimates RTTs to all the receivers and uses the TCP throughput characterization equation derived in [19] to calculate the throughput that a TCP connection would attain under the same condition (i.e., with the same RTT, packet loss rate, and so on). The sender then selects, among all the receivers, the one that attains the minimum throughput as the *Acker*. Since a sender is responsible for *Acker* selection, it has to handle all the NAKs from the receivers, and hence is subject to the NAK implosion problem.

Rate-based Approaches:: The rate-based approaches can be further classified into two categories: those that adjust their sending rate in an additive increase/multiplicative decrease fashion as in TCP [20], [26] and those [18], [28], [27] that adjust the sending rate in compliance with the TCP throughput characterization equations derived in [17], [19].

The rate adaptation protocol (RAP) [20] falls in the first category. In RAP, the packet transmission rate is reduced by half when packet loss is detected and is increased by a small amount in the absence of packet loss in one RTT. Sisalem *et al.* [23] proposed to use a modified RTP/RTCP report transmitted between the receiver and the sender to estimate the RTT, the packet loss rate, and the bottleneck link bandwidth. Based on these estimated parameters, the scheme then adjusts the sending rate using an additive increase and multiplicative decrease method. Although simple and feasible, this scheme contains several tunable parameters that must be determined by the user. Both of the above schemes focus on unicast, and it is not clear whether or not, and how, they can be extended to multicast.

In [15], [26], [9], congestion control is realized by transmitting data in a layered manner. Video data is encoded into a number of layers that can be incrementally combined to provide progressive refinement. The receivers of each layer constitute a multicast group. The number of layers that a receiver subscribes to depends on its perceived packet loss rate and outcomes of its join experiments. The work in [26] achieves TCP-like additive increase and multiplicative decrease congestion control by using strict time limits on when a receiver may join/leave a group. Although the above layered multicast approaches [15], [26] in general perform well, they have been shown in [12] to exhibit significant instability and several pathological behaviors, such as slow convergence, high loss rate, conservative or aggressive behaviors in some cases as compared to TCP. In [9], the authors argue that join/leave coordination is difficult to achieve in [15], [26] if the tree topology is not known. Hence, they develop an algorithm that uses the tree topology and the loss information to determine the optimal subscription bandwidth for each receiver. The major drawbacks of this approach are that it is centralized and that the multicast tree topology and the bandwidth available on each link may not be readily available in reality.

In the second category of rate-based approaches, the sending rate is adjusted with respect to the measured loss rate and RTTs, using the TCP characterization equation derived in [17], [19], [16]. In particular, the protocol proposed in [25] is based on the TCP throughput characterization that does not take into account of timeouts (reported in [16], [17]), while those in [18], [27] are based on the TCPs throughput characterization that does (reported in [19]). The scheme in [25] relies on the notion of layering. A receiver estimates its RTT and packet loss rate, and calculates the TCP-friendly rate. Based on the calculated rate, each receiver then determines dynamically whether or not to join/leave certain layers.

Padhye *et al.* [18] proposed the CMTCP protocol in which a more accurate TCP characterization equation reported in [19] is used to adjust the sending rate. Although CMTCP has been validated by simulation and real-world experimentation to be TCP-friendly, authors of CMTCP also observed the following difficulties: (i) measuring packet loss rates accurately is not an easy task; (ii) the re-computation interval, M, over which on-line estimated parameters are updated has an impact on the performance and has to be fine tuned; (iii) CMTCP behaves more aggressively than TCP under the small RTT case.

TFMCC [28] extends the TCP-friendly TFRC [4] protocol from unicast to multicast. In TFMCC, receivers es-

timate their packet loss rates and RTTs between them and the sender and calculate TCP-friendly transmission rates. The rates are fed back to the sender. If a receiver sends back a rate which is lower than the sender's current rate, the sender will immediately reduce its sending rate to that rate. Feedback suppression is achieved by using a random feedback timer method. The major pitfall of TFMCC is that it is difficult to set timers to effectively suppress feedbacks while responding to congestion promptly.

V. Simulation Results

We have implemented *RACCOOM* on *ns-2*, and conducted a simulation study to (i) validate the proposed design in terms of fairness, TCP-friendliness, capability to deal with persistent congestion and membership change; and to (ii) measure the overhead for ACK aggregation.

We consider network topologies with a single bottleneck link (dumbbell-like) , multiple bottleneck links (e.g., Fig. 7), and arbitrary network topologies (e.g., Figs. 9). Also, we use an assortment of traffic sources (mainly infinite-duration TCP, finite-duration TCP, and on-off UDP sources).

In spite of quite a number of system parameters and algorithm parameters 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 particular, the performance of RACCOOM is rather insensitive to the values of k_1 and k_2 (as long as they are appropriately chosen to drive the system into equilibrium). In all the experiments reported below, k_1 and k_2 are fixed at 4 and 0.25, respectively. The initial sending rate of a *RACCOOM* session is set to 5K bytes/second. Packets are of length 500 bytes. The values of α and β are on-line adjusted (using Eq. (10)) in experiments in which TCP-friendliness is the performance measure. Designated receivers are selected using the algorithm reported in [10].

The performance metrics of interest are the "friendliness ratio," F, defined as $F = \frac{r_{RACCOOM}}{r_{TCP}}$, where $r_{RACCOOM}$ and $r_{\rm TCP}$ are the attainable throughput of a *RACCOOM* session and a TCP connection under competition; and the fairness ratio, F_r, of a *RACCOOM* connection to the other competing *RACCOOM* connections, defined as $F_r = \frac{r_i}{\min_j r_j}$, where r_i is the sending rate of the *RACCOOM* connection under consideration and $\min_j r_j$ is the minimum value of the sending rates of the other competing *RACCOOM* connections.

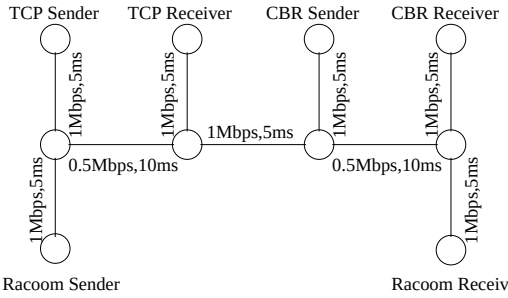


Fig. 7. A network topology with multiple bottleneck links used in the simulation

Fairness amongst RACCOOM *connections::* In the first and second experiments, we study fairness amongst *RACCOOM* connections with different RTTs. In the first experiment, the single-bottleneck network topology is used. The bottleneck link is shared among 10 *RACCOOM* connections and has a capacity of 0.5 Mbps, a delay of 20 ms, and a buffer of size 10 Kb. The links between a sender and the left router and between a receiver and the right router have a capacity of 1 Mbps and a delay that ranges from 5 to 140 ms in different simulation runs. A total of 10 simulation runs are conducted. In the ith ($1 \leq i \leq 10$) simulation run, the delays between senders/receivers and their attached routers are evenly distributed in $[5, 15 * i - 10]$ ms. Thus, the maximum value, *MaxRTT*, of the RTTs among the 10 connections is $\frac{2(15*i-10)+20}{2*5+20} = i$ times of the minimum value, *MinRTT*, of RTTs. The values of α and β are set to 2 and 6,

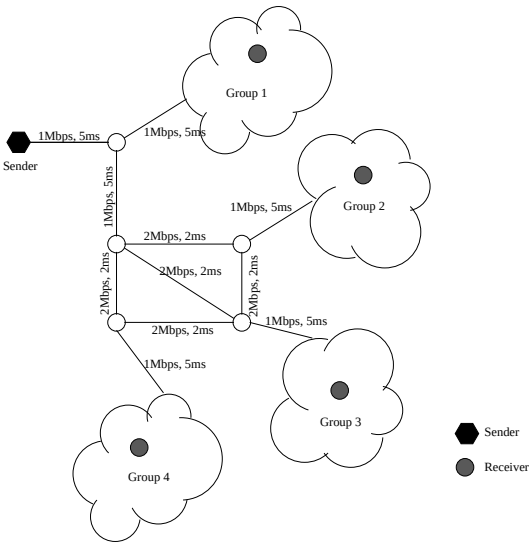
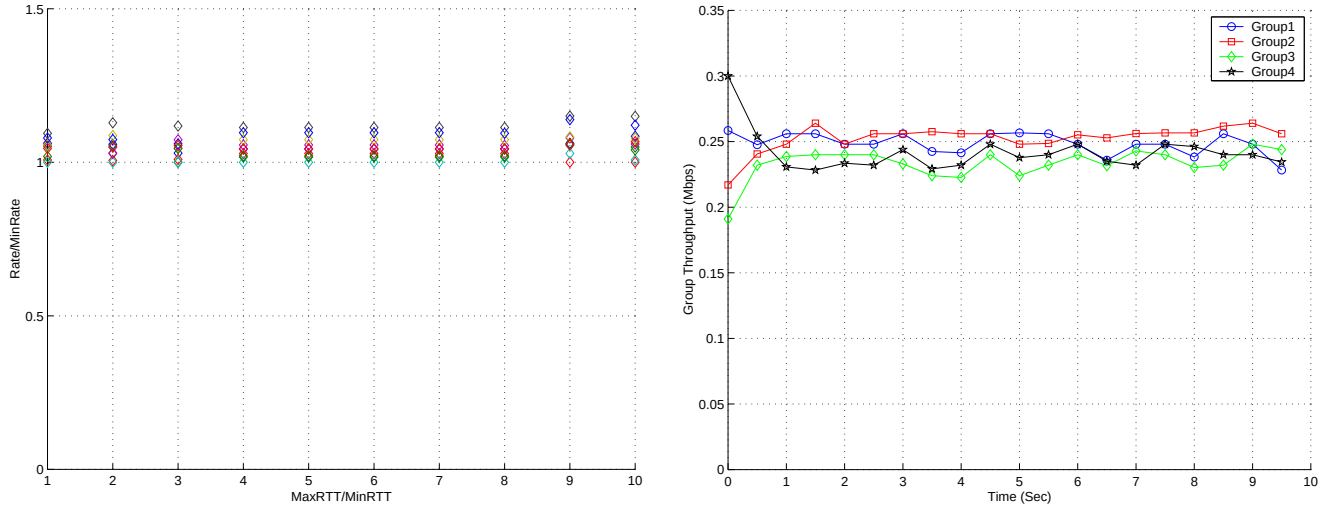


Fig. 8. An arbitrary network topology used in the simulation.



(a) single bottleneck topology

(b) arbitrary topology

Fig. 10. The performance of *RACCOOM* in terms of F_r.

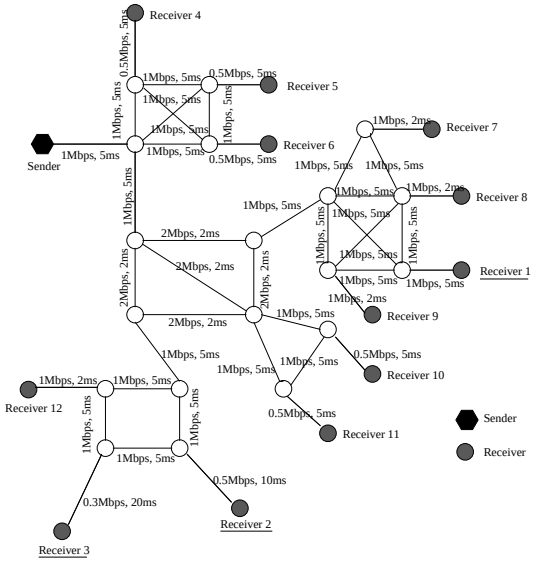


Fig. 9. Another arbitrary network topology used in the simulation.

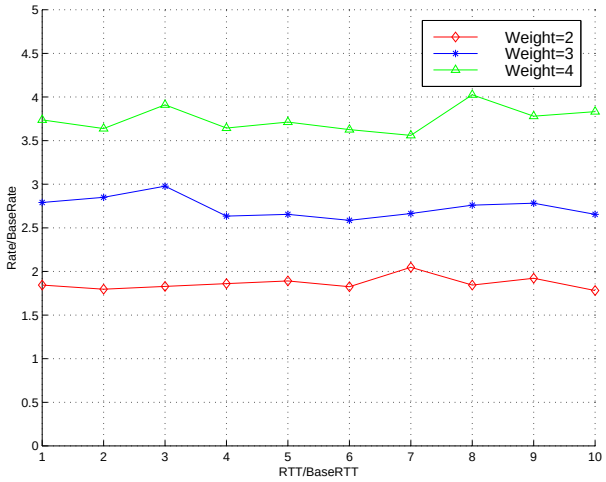


Fig. 11. The performance of *RACCOOM* in terms of weighted fairness.

respectively. Fig. 10 (a) gives the simulation results in terms of F_r with respect to $\frac{\text{MaxRTT}}{\text{MinRTT}}$. As shown in Fig. 10 (a), the values of F_r's ranges from 1.0 to 1.15 in all the simulation runs, i.e., the bottleneck link bandwidth is shared, independently of RTT, among the 10 *RACCOOM* connections in a fair manner.

The second experiment is conducted in the arbitrary network topology given in Fig. 8: there are four multicast groups, each with 100 receivers. Four *RACCOOM* multicast sessions are established between the sender and the receivers. Four groups share the bottleneck link of 1 Mbps. Fig. 10 (b) gives the sending rate attained by the four sessions in a period of 10 seconds. Consistent with the analysis in Section III, after an initial transient time, each of the *RACCOOM* sessions approximately attains a throughput of 0.25 Mbps. Both the first and second experiments validate the fairness analysis in Section III.

Weighted fair share amongst RACCOOM *connections::* In the third experiment, we evaluate *RACCOOM* in terms of its weighted fairness capability. The

single-bottleneck network topology is used. The bottleneck link has the same attributes as in the first experiment, and is shared by 31 *RACCOOM* connections. One *RACCOOM* connection is used as the reference connection (with $\alpha = 2$, $\beta = 6$, and $E = \frac{\alpha+\beta}{2} = 4$), and the other 30 *RACCOOM* connections are evenly grouped into 3 groups. The connections in the first, second, and third group are assigned a weight of 2, 3, 4, respectively, and the (α, β, E) values used for the connections in the first, second, third group are $(6, 10, 8)$, $(10, 14, 12)$, and $(14, 18, 16)$, respectively.

As shown in Fig. 11, the ratio of the sending rate of a group-i connection to that of the reference connection is approximately equal to the weight assigned to the connection, and is rather insensitive to RTT changes. This validates the analysis (Section III) that with the expected queue length along the target path appropriately assigned, *RACCOOM* can achieve weighted fairness among competing connections.

TCP-friendliness:: In the fourth and fifth experiments, we evaluate *RACCOOM* in terms of TCP friendliness. In the fourth experiment, the single-bottleneck network topology is used. The bottleneck link has the same attributes as in the first experiment, and is shared by 1 *RACCOOM* connection (with $k_1 = 2$, and $k_2 = 0.25$) and 10 TCP connections. The values of α and β used in *RACCOOM* are online adjusted according to Eq. (10). Three simulation runs are conducted. The values of RTTs are the same for all the connections and are set to 40, 60, and 80 ms in the three simulation runs, respectively.[7] Fig. 12 (a) gives the ratio of the bandwidth attainable by the *RACCOOM* connection to that by each TCP connection. The ratio is very close to 1 for all the TCP connections in all three simulation runs. This shows that *RACCOOM* achieves TCP-friendliness, for a wide variety of RTT values.

To verify whether or not *RACCOOM* sessions still exhibit TCP-friendliness in arbitrary network topologies, we conduct simulation in an arbitrary network topology given in Fig. 8 in the fifth experiment: one *RACCOOM* multicast session is established between the sender and each multicast group. The group size of each multicast group varies from 5 to 100. In addition, one TCP connection is established between the sender and the receiver that lies on the target path in each multicast group. All the connections share the bottleneck link of 1 Mbps. Fig. 12 (b) gives the ratio of the sending rate of a *RACCOOM* session to that of the corresponding TCP connection of group 1. The ratios ranges from 0.8 to 1.5, which indicates that *RACCOOM* can achieve TCP-friendliness in an arbitrary topology with numerous receivers.

[7] As the RTT value from a host at OSU to a host at UCSB is approximately 60 ms, we consider the range of $[40, 80]$ in which RTT varies adequate.

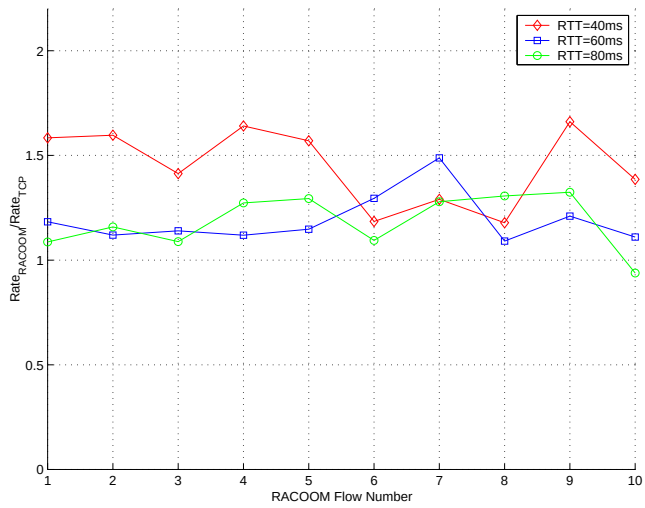
Receiver	BW(Mbps)	RTT(ms)
1	1	54
2	0.5	64
3	0.3	94
4,5,6	0.5	30
7,8,9,12	1	48
10,11	0.5	44

TABLE I

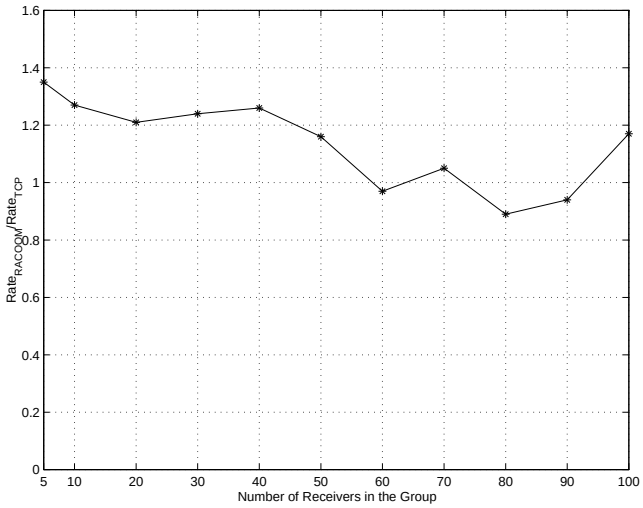
THE RTT VALUES AND THE BANDWIDTH OF BOTTLENECK LINKS BETWEEN THE SENDER AND THE RECEIVERS.

Performance in the case of membership change:: In the sixth experiment, we study how *RACCOOM* responds to membership change. We use the arbitrary network topology given in Fig. 9 (with the link delay and the link bandwidth labeled in the figure). There is one *RACCOOM* multicast session (with $\alpha = 2$ and $\beta = 6$), with one sender and 400 hundred receivers. Among the receivers, 12 representative receivers that are on potential target paths are labeled in the figure. The RTT values and the bandwidth of the bottleneck links between the sender and the receivers are listed in Table. I. Initially only receivers 1, and 4–12 are in the multicast group, and the on-tree path from the sender to receiver 1 incurs the largest RTT value and is identified as the target path. Hence, the throughput the *RACCOOM* session can attain is constrained by the bandwidth of the bottleneck link (1 Mbps) on the target path. At time instant 3, receiver 2 joins the group and introduces a new target path with RTT$= 64$ ms and the bottleneck link bandwidth of 0.5 Mbps. At time instant 8, receiver 3 joins the group and introduces a new target path with RTT$= 94$ ms and the bottleneck link bandwidth of 0.3 Mbps. Finally, at time instant 16 and 24, receiver 2 and 3 leave respectively.

As shown in Fig. 13 (a), the sending rate of the sender is initially 1 Mbps, reduced to 0.5 Mbps when receiver 2 joins the multicast group (at time instant 3), further reduced to 0.3 Mbps when receiver 3 joins the group (at time instant 8), increased to 0.5 Mbps when receiver 3 leaves (at time instant 16), and finally increased to 1.0 Mbps when receiver 2 leaves (at time instant 24). This demonstrates the capability of *RACCOOM* in keeping track of the correct target path in the case of membership change.

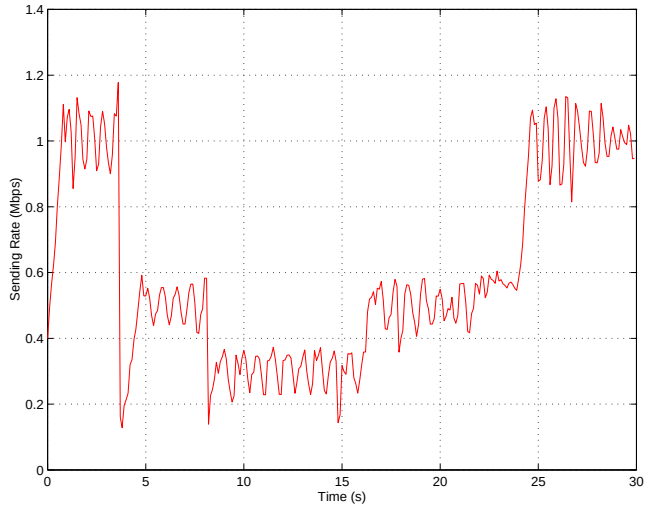


(a) single bottleneck topology

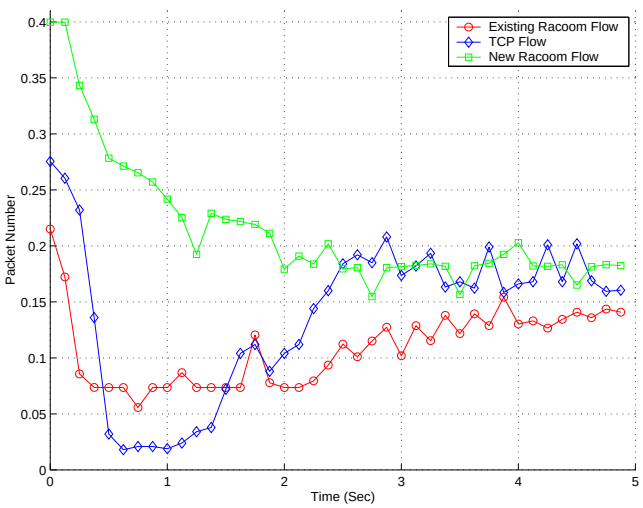


(b) arbitrary topology

Fig. 12. The performance of *RACCOOM* in terms of TCP-friendliness.



(a) capability to deal with membership change



(b) capability to deal with persistent congestion

Fig. 13. The performance of *RACCOOM* in terms of (a) the capability to deal with membership change in the arbitrary topology and (b) the capability to deal with persistent congestion.

Performance in the case of persistent congestion:: In the seventh experiment, we study the effect of persistent congestion on the performance of *RACCOOM*. The single-bottleneck network topology is used. The topology has the same attributes as in the first experiment. Initially the bottleneck link is shared, and fully utilized, by a TCP connection and a *RACCOOM* connection. At time instant 0, a new *RACCOOM* connection is established. Since the bottleneck link is fully utilized by existing connections, the new *RACCOOM* connection will over-estimate the value of RTT_{min} under the persistent congestion condition. Fig. 13 (b) depicts the throughput attainable for each of the three connections for a time period of 5 seconds. Initially the new *RACCOOM* connection starts with a high sending rate because of the over-estimated value of RTT_{min}. As a result, congestion occurs and packets are dropped at the bottleneck

link. When both the existing and new *RACCOOM* senders detect packet loss, they reduce their sending rates by half, but only once every RTT. In contrast, the TCP connection repeatedly reduces its congestion window by half, and at certain point, even shuts down the congestion window. After a transient period of approximately 2.8 seconds, the sending rates of the three connections stabilize at approximately 0.16 Mbps and the bandwidth of the bottleneck link is evenly shared by the three connections.

Performance in the existence of multiple bottleneck links:: In the eighth experiment, we analyze how the throughput attainable by a *RACCOOM* connection is affected when the connection traverses more than one bottleneck link. The multiple-bottleneck network topology shown in Fig. 7 is used. Both bottleneck links have a capacity of 0.5 Mbps, a latency of 10 ms, and a buffer of 10 Kb. The capacity and latency of other links are 1 Mbps and 5 ms, respectively. One bottleneck link is shared by a *RACCOOM* connection and a TCP connection (that lasts for 20 seconds), and the other is shared by the same *RACCOOM* connection and a UDP CBR connection (that sends at 0.4 Mpbs and lasts for 10 seconds). The values of α and β of the *RACCOOM* connection are on-line adjusted. As shown in Fig. 14 (a), because of the existence of the non-responsive UDP CBR connection in the time interval of [0,10 sec], the *RACCOOM* connection only attains the throughput of 0.1 Mbps (which is the bandwidth left on the second bottleneck link). The TCP connection that traverses the other bottleneck link thus attains the throughput of 0.4 Mbps. When the UDP CBR connection terminates at time = 10 seconds, there is only one bottleneck link on which the *RACCOOM* connection shares the bandwidth with the TCP connection in a TCP-friendly manner (each attains a throughput of 0.25 Mbps). When the medium-duration TCP connection terminates at time = 20 seconds, the *RACCOOM* connection responds by capturing all the bandwidth on the bottleneck link.

Ack aggregation for scalability:: In the ninth experiment, we measure how many ACK messages arrive at a *RACCOOM* sender. The arbitrary network topology given in Fig. 9 is used. The simulation setup is the same as that in the sixth experiment, except that both the router-assisted and designated receiver-assisted ACK aggregation approaches are used in two simulation runs. Fig. 14 (b) gives the number of ACK messages received versus the number of data packets sent. (The results obtained under both aggregation approaches are indistinguishable from each other and only one curve is depicted.) Ideally (one ACK for each data packet sent), the result should be a 45 degree straight line. As shown in Fig. 14 (b), the result even falls below the 45 degree line, due the the round trip delay between sending of a data packet and receipt of its corresponding acknowledgement. This also shows that the ACK aggregation mechanism used in *RACCOOM* indeed aggregates ACK messages effectively and prevents the occurrence of ACK implosion.

VI. Conclusion

We have presented in this paper a rate-based congestion control scheme, *RACCOOM*. In the absence of packet loss, a *RACCOOM* sender adjusts its sending rate in a TCP-Vegas fashion, based on the congestion status of the on-tree path with the largest RTT (called the *target* path). In case of packet loss, a *RACCOOM* sender then responds by reducing its sending rate by half. An ACK aggregation method is judiciously devised to prevent ACK implosion and yet to provide the sender with a simple but comprehensive view of congestion conditions in the multicast tree. *RACCOOM* also achieve (weighted) fairness among competing connections by exploiting feedback control theory and appropriately selecting the parameters used in the rate adjustment mechanism. On the other hand, if TCP friendliness is the performance criterion, then a simple iterative approach can be used to on-line adjust the parameters α and β so as for a *RACCOOM* session to exhibit TCP-friendliness.

Simulation experiments indicate that *RACCOOM* connections can achieve, irrespectively of the RTTs of individual connections, TCP-friendliness, can handle membership/network traffic changes, can deal with persistent congestion, and can achieve (weighted) fairness among competing connections with different RTTs.

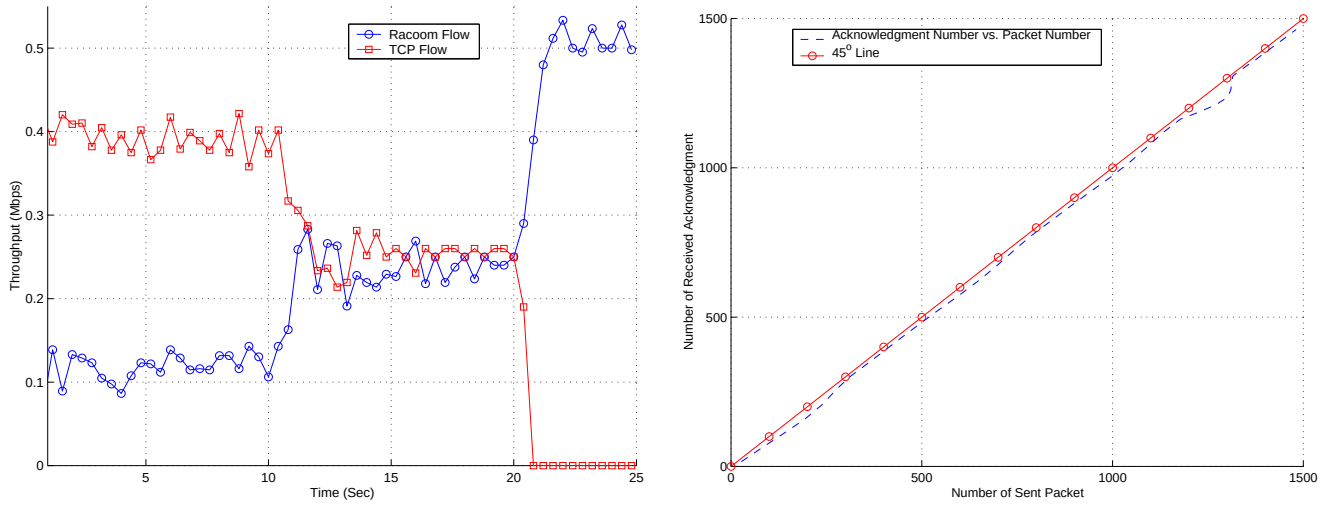
We have identified several avenues for future work. We plan to prototype *RACCOOM* on FreeBSD and conduct experiments over Internet 2. Since both vBNS and Abilene currently run PIM-SM [5]/MBGP [1]/MSDP [6] as the inter-domain multicast routing protocol and source discovery protocol, we will study how to implement *RACCOOM* on top of these protocols.

References

[1] T. Betes, R. Chandra, D. Katz, and Y. Rekhter, "Multiprotocol extensions for BGP-4," IETF RFC 2283, February 1998.

[2] S. Bhattacharyya, D. Towsley, and J. Kurose. The loss path multiplicity problem in multicast congestion control. *Proc.of IEEE INFOCOM'99*, New York, NY, March 1999.

[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: end to end congestion avoidance on a global internet. *IEEE JSAC*, 13/8, pp. 1465–80, October 1995.



(a) existence of multiple bottleneck links (b) ACK aggregation

Fig. 14. The performance of *RACCOOM* (a) in the existence of multiple bottleneck links and (b) in terms of ACK aggregation.

[4] S. Floyd, M. Handley, J. Padhye and J. Widmer. Equation-based congestion control for unicast applications. In Proc. of *ACM SIGCOMM'2000*, 2000.

[5] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast sparse-mode (PIM-SM): protocol specification. IETF RFC 2362, June 1998.

[6] D. Farinacci, Y. Rekhter, P. Lothberg, H. Kilmer, and J. Hall. Multicast source discovery protocol (MSDP). IETF Internet Draft, June 1998.

[7] S. J. Golestani. Fundamental observations on multicast congestion control in the Internet. *Proc. of IEEE INFOCOM'99*, New York, NY, March 1999.

[8] H. W. Holbrook and D. R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. *Proc. of ACM SIGCOMM'99*, 1999.

[9] S. Jagannathan, K. Almeroth, and A. Acharya. Topology Sensitive Congestion Control for Real-Time Multicast. *Proc. of NOSSDAV'2000*, June 2000.

[10] M. Kadansky. Reliable multicast tree-building techniques. RMRG Meeting, July 1998, UCL, London.

[11] K.-W. Lee, S. Ha and V. Bharghavan. IRMA: a reliable multicast architecture in the Internet. *Proc. of IEEE INFOCOM'99*, New York, NY, March 1999.

[12] A. Legout, and E. W. Biersack Pathological Behaviors for RLM and RLC. *Proc. of NOSSDAV'2000*, June 2000.

[13] B.N. Levine, S. Paul, and J.J. Garcia-Luna-Aceves. Organizing multicast receivers deterministically according to packet-loss correlation. *Proc. of ACM Multimedia 98*, September 1998.

[14] J. Lin, S. Paul, K. Sabnani, and S. Bhattacharyya. A reliable multicast transport protocol(RMTP). *IEEE JSAC*, 15, April 1997.

[15] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast *Proc. of ACM SIGCOMM'96*, 1996

[16] M. Mathis, J. Semke, J. Mahdavi, T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communication Review*, Vol. 27, No. 3, July 1997.

[17] T. J. Ott, J. H. B. Kemperman and M. Mathis. The stationary behavior of ideal TCP congestion avoidance. *Proc. of IEEE INFOCOM'99*, New York, NY, March 1999.

[18] J. Padhye, J. Kurose, D. Towsley and R. Koodli. A TCP-friendly rate adjustment protocol for continuous media flows over best effort networks. UMass-CMPSCI Tech. Report TR 98-04, October 1998.

[19] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. UMass-CMPSCI Tech. Report TR 98-008, Feb 1998.

[20] R. Rejaie, M. Handley and D. Estrin. RAP: an end-to-end rate-based congestion control mechanism for realtime streams in the Internet. *Proc. of IEEE INFOCOM'99*, New York, NY, March 1999.

[21] L. Rizzo. pgmcc: a TCP-friendly single-rate multicast congestion control scheme. *Proc. of ACM SIGCOMM'2000*, 2000.

[22] I. Rhee, N. Ballaguru, G. N. Rouskas. MTCP: scalable TCP-like congestion control for reliable multicast. *Proc. of IEEE INFOCOM'99*, New York, NY, March 1999.

[23] D. Sisalem, and H. Schulzrinne. The loss-delay adjustment algorithm: a TCP-friendly adaptation scheme. *Proc. of NOSSDAV'98*, July 1998.

[24] R. Talpade, and M. H. Ammar. Single Connection Emulation: An Architecture for Providing a Reliable Multicast Transport Service. *Proc. of ICDCS'95*

[25] T. Turletti, S. Parisis and J. Bolot. Experiments with a layered transmission scheme over the Internet. Tech. report RR-3296, INRIA, France, 1996.

[26] L. Vicisano, L Rizzo and J. Crowcroft. TCP-like congestion control for layered multicast data transfer. *Proc. of IEEE INFOCOM'98*, San Francisco, CA, March 1998.

[27] B. Whetten, J. Conlan. A rate based congestion control scheme for reliable multicast. Technical White Paper, GlobalCast Communications, October 1998.

[28] J. Widmer, and M. Handley. Extending equation-based congestion control to multicast applications. In Proc. of *ACM SIGCOMM'2001*, 2001.