A Receiver Based Single-Layer Multicast Congestion Control Protocol for Multimedia Streaming*

Miguel Rodríguez-Pérez, Manuel Fernández-Veiga, Sergio Herrería-Alonso, Andrés Suárez-González, and Cándido López-García

E.T.S.E. Telecomunicación
Campus universitario s/n
36 200 Vigo, Spain

Abstract. The transmission of multimedia content in a multicast environment is still a topic of active research. New congestion control protocols that can support the needs of real time transmission, while keeping the current Internet stability, are needed. While there already exist some protocols suitable for multimedia transmission in unicast communications, and there are also protocols capable of transmitting data to a large number of receivers simultaneously, our protocol is designed for achieving both goals at the same time. In this paper we present a novel protocol designed for multimedia streaming in multicast networks. Our simulation results show that the protocol is compatible with TCP flows, thus preserving the network stability, and is suitable for real-time multimedia transmissions, as it has low oscillations in its throughput and imposes no additional network delays that could increase the latency.

Keywords: Multicast, Congestion Control, Multimedia

1 Introduction

Multimedia streaming across the Internet to a large and disperse number of receivers has still to improve before being safely deployed. At the network level, the essence of the problem lies in devising a bandwidth-efficient transport protocol which does not jeopardize the stability of the current Internet.

In designing a TCP-compatible congestion control algorithm for multicast streaming, we have identified the following set of desirable features:

Smoothness. It is well known that the abrupt changes in the throughput (window) of a TCP connection, due to the loss of even a single packet, can severely degrade the subjective quality of a multimedia flow [1,2,3,4]. Therefore, it is imperative that any candidate protocol adjusts its transmission

* This work was supported by the project "TIC2 000–1 126 of the Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica" and by the grant PGIDT01PX132202PN of the "Secretaría Xeral de I+D da Xunta de Galicia."

N. Mitrou et al. (Eds.): NETWORKING 2004, LNCS 3042, pp. 550–561, 2004.

rate at a slow pace. Slowly-responsive algorithms also allow the use of smaller buffers at the receivers which, in turn, help to reduce overall latency.

Low latency. This requirement is not as much important for one-way communications as it is for two-way communications. So, although not strictly needed, it is needed for the protocol to support interactive applications.

Scalability. The protocol should be deployable in multicast networks with a large number of receivers. This point poses some interesting design issues, especially the avoidance of the catastrophic feedback implosion phenomenon [5]. It is critical to find a way to collect enough state information about the receivers without saturating the network with their feedback messages.

Stability. Network conditions vary dynamically. After any change, the protocol must eventually reach a steady state where all other goals are preserved.

Intra-protocol fairness. The protocol must be fair to other instances of itself. Informally, if two sessions are experiencing similar end-to-end network conditions they should obtain equitable bandwidth.

Inter-protocol fairness. No other protocols should be mistreated when they compete for bandwidth along a network path or link. This compatibility requirement is as essential as the intra-protocol fairness, because it guarantees safe deployment on the network. In this paper, we limit ourselves to protocols fair against TCP, since this is the predominant traffic in the Internet [6, 7].

So as to fulfill the goals stated above, we ought to make some choices about the use of different techniques. The last years have witnessed important advances in this area, resulting in two broad approaches: the *multiple-layer* proposals and the *single-rate* approach. While it is better to use a multiple-layer approach if best-quality delivery to a very different set of receivers is the concern [8], single-rate protocols are a better fit for keeping the sender and the encoding of the data simpler [2,9]. In addition, single-rate protocols are generally easier to understand and analyze because of their similarity with unicast protocols. Hence, in this paper we focus on a single-rate algorithm.

There exist two main strategies aimed to avoid network congestion suffered with the single-rate multicast streaming protocols, both borrowed from unicast streaming solutions and based on the robust congestion control mechanisms embedded in TCP. One of the options is to rely on an equation-based algorithm, like in [2], that, by using an approximate throughput formula for TCP as a non-linear control rule, tries to track the same transmission rate that a real TCP connection would have between the two endpoints. However the equation-based protocols react too conservatively to changes in the network conditions and do not efficiently use all the available bandwidth in dynamic conditions [10]. The alternative pursued by TCP *emulators* is to reproduce directly the behavior of TCP. In this case, and considering a multicast transmission paradigm, the main problem is getting enough feedback information (acknowledgments) from all receivers without flooding the sender. One possible solution is using complicated algorithms for aggregating feedback from the receivers like in [11]. Another possibility, chosen in our work, is to use a representative of all the receivers [9,

Fig. 1. Main elements present in a VLMCC transmission.

12]. This schema does not necessarily need any help from routers in order to be scalable and avoids completely the threat of *feedback implosion*. Furthermore, unlike [9,12] where non-representative receivers send a NACK packet whenever they detect a packet loss, in our proposed protocol these receivers only send feedback when they consider that they should take the role of the representative host, reducing, even more, the feedback traffic.

Another problem we faced in our design was devising a way for the protocol to be TCP-fair. To this end, we have preferred to emulate the TCP behavior between the sender and a representative host. This way, we can trust that, if the representative is well chosen, i.e., if it is the host with the worst connection among all, then the protocol will be TCP-fair along all the links. The specific TCP implementation chosen is TCP-Vegas [13], hence the name of our proposed protocol: VLMCC. TCP-Vegas was chosen for two main reasons: (1) It usually exhibits small variations in its sending rate, yielding a smooth throughput in the transmission; (2) With TCP-Vegas it is easier for the receivers to analyze the network conditions in a given instant. If the sending rate fluctuates amply, like in other TCP implementations, it becomes more difficult to know what the long term sending rate is, and this would force the receivers to delay their responses to congestion until a reliable measure of the sending rate had been obtained.

The rest of this paper is organized as follows. Section 2 presents the protocol architecture and its behavior. In Section 3 simulation results are shown. Our conclusions are exposed in Section 4.

2 Protocol Description

2.1 High Level Overview

VLMCC works by emulating a TCP-Vegas connection between the *sender*, represented with a triangle in Fig. 1, and the host experiencing the worst network conditions, called the *leader*. The rest of the hosts, the *followers*, are passive most of the time and they simply monitor the network conditions, asking the sender to become the new leader if they detect that their conditions are worse than those of the *leader*.

Unlike PGMCC [9] where the sender is in charge of detecting hosts in worse conditions than those the leader is experiencing, in VLMCC the receivers themselves are able to detect this situation and report it to the sender. This frees the

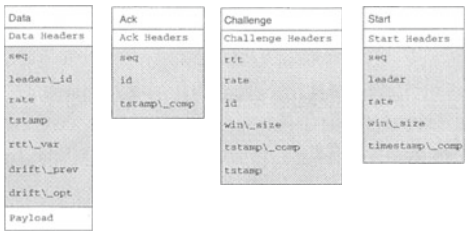


Fig. 2. Information interchanged between the sender and the rest of the hosts in the transmission. The area in grey holds the data needed by VLMCC.

server of some load and completely avoids the problem of the sender not electing the worst receivers because of NAKs suppression.

We have depicted in Fig. 2 the different packets interchanged between the sender and the rest of the nodes participating in the transmission. *Data* packets and *Start* packets are sent by the sender, *Ack* packets are sent by the session leader and *Challenge* packets can only be sent by follower nodes. In the following sections, we will show what the information conveyed in those packets is for.

2.2 The Sender-Leader Pair

The Sender-Leader pair is responsible for emulating the behavior of a TCP-Vegas connection running between those two nodes. For the sake of this discussion, we will assume that the leader has already been elected.

To accomplish their mission the Sender-Leader Pair uses the *Ack* packet and three fields of the *Data* packet (cf. Fig. 2): `seq`, `leader_id` and `tstamp`. The other fields are just information for the rest of the nodes. `seq` is a sequential number used to differentiate the packets. `leader_id` is the address of the current leader, and it is used by the receivers to know if they have been chosen as the leader responsible to emit an acknowledgment for the packet. The `id` field in the *Ack* packet is there just to prevent errors and contains the address of the node that emits the *Ack*.

The sender acts like a simplified TCP-Vegas server that does not provide any guarantee about delivery of the packets neither has to bother with retransmissions. In contrast, the job of the leader is much easier, for it only has to answer with an ack to every packet it receives. Using the variations in the RTT to detect incipient congestion, as TCP-Vegas does, VLMCC is able to react lowering its transmission rate before packet losses are likely to happen, avoiding drastic reductions in the window size. Therefore, to properly emulate the TCP-Vegas behavior, the sender must maintain a congestion window (W), and estimations of both the RTT and its variance ($\widehat{rtt}$ and $\hat{\sigma}^2_{\text{rtt}}$ respectively). Both $\widehat{rtt}$ and $\hat{\sigma}^2_{\text{rtt}}$ are estimated using exponential moving averages according to (1) and (2), where g and g_{σ^2} are gains and rtt is the value of the RTT observed by the last acknowledged packet. We have chosen, rather empirically, $g = g_{\sigma^2} = 0.1$.

$$\widehat{rtt} \leftarrow (1 - g)\,\widehat{rtt} + g \cdot rtt \tag{1}$$

$$\hat{\sigma}^2_{\text{rtt}} \leftarrow (1 - g_{\sigma^2})\,\hat{\sigma}^2_{\text{rtt}} + g_{\sigma^2}\left|\widehat{rtt} - rtt\right| \tag{2}$$

For measuring the *rtt*, the sender could just store the time it sends every packet and compare it with the time it receives its corresponding ack, but in doing so, the time spent by the receiver generating the ack would be part of the measured *rtt*. To avoid this bias the sender timestamps the *Data* packet (`tstamp` field) and the leader echoes a corrected value back in the *Ack* packet (`tstamp_comp` field). Instead of just copying the `tstamp` into `tstamp_comp`, leaders add the time spent by the receiver processing the packet to it. This way, the sender obtains a more accurate measure of *rtt* that accounts only for the time the packet actually spent in the network.

Every $\widehat{rtt}$ the sender adjusts its congestion window according to the variations observed in the RTT. To this end, it compares the obtained throughput ($packetsInTravel/\widehat{rtt}$), in packets per RTT, with the maximum expected of ($W/base\_rtt$), where *base_rtt* is the minimum *rtt* measured in the transmission. If the difference is below a certain threshold α,[1] the sending rate can still be safely increased, and so W is increased in one packet. When the difference is considered too high, i.e., when it is above a certain level β, the connection is starting to cause congestion, and W is decreased in one packet. In any other case, W is kept constant.

The sender also modifies W whenever packet losses are detected (packets whose acks do not arrive in $\widehat{rtt} + 4\,\hat{\sigma}^2_{\text{rtt}}$ seconds after they have been sent[2]) during a RTT period. When such a loss is detected, the value of W is halved. Other packet losses occurring in the same RTT would be ignored, as the decrease in the congestion window size has not taken effect for those packets yet.

With all of this taken into account, a VLMCC sender can send packets to the network provided all of the following conditions hold true:

1. The sending application has pending packets to send.
2. The number of packets in travel is less than the congestion window size.
3. Some time has passed since the previous burst of packets was transmitted. This condition is a consequence of the fact that TCP-Vegas and VLMCC try to avoid sending long bursts of packets to the network. In our protocol, we have limited the burst length to just two packets, after which the sender has to wait the delay dictated by (3) before sending the next packet. In any case, a full window of packets can still be sent in just half the time of an RTT.

$$delay = \widehat{rtt} \cdot \frac{MaxSegment}{W} \tag{3}$$

[1] The difference is first normalized dividing $\widehat{rtt}$ over *base_rtt*, so α and β can be measured in bytes or packets and not in bytes per second or packets per second. This lets the configuration parameters do not depend on the physical capabilities of the underlying network, such as the links bandwidth or their length.

[2] This is the same value as the TCP retransmission timeout [6].

2.3 The Followers

The followers are nodes who are experiencing better network conditions than the *leader*. So, their job is to monitor network conditions and, provided such conditions worsen, warn the sender requesting to replace it. We call this *challenging the leader*, and the procedure involves sending a *Challenge* packet.

The tricky part is, of course, monitoring the network conditions, because the followers are forbidden to send any regular feedback as that would render the protocol unscalable. So, the followers estimate the network conditions by calculating the congestion window size a TCP-Vegas connection would have in use under the same circumstances.

Followers apply the same algorithm described in Section 2.2, but with one additional difficulty: since they cannot send regular data to the *sender*, they cannot know directly the value of *rtt*. The solution proposed here consists on asking the sender for a little help and not calculate *rtt* directly, but rather its variation (Δ_{rtt}) between two consecutive packets.

When a node joins a transmission it must get a first estimation of *rtt*. For this it sends a special timestamped packet to the sender that is always answered. Receivers use a *Challenge* packet for this, but only two fields are filled: `id` and `tstamp`. `id` carries the identity of the node that sends the *Challenge* while `tstamp` is used by the receiver to measure *rtt*. The answer comes in the form of a *Start* packet, that carries the following info: the `seq` number of the next expected packet, `leader` which is the address of the leader receiver, `rate` that is the current sending rate of the sender, `win_size` as the current congestion window size and finally `timestamp_comp`, which is used to measure *rtt*.

After this initial estimation, the follower adjusts *rtt* according to

$$rtt \leftarrow rtt + \Delta_{\text{rtt}} \tag{4}$$

every time two consecutive packets are received. With this *rtt* value, the estimation $\widehat{rtt}$ can be computed using (1). But how is Δ_{rtt} calculated? The sender just provides the needed information in every packet by inserting the time since the previous packet was sent in the `drift_prev` field of *Data* packets. So

$$\Delta_{\text{rtt}} = now - (lastArrivalTime + drift\_prev). \tag{5}$$

One remarkable side benefit of calculating Δ_{rtt} that way, is that only takes into account the variations suffered in the direction data is travelling. This makes the protocol robust against other connections causing congestion in the return path. If VLMCC reacted to congestion in the return path, not only it would not alleviate the congestion, but it would also affect the performance adversely.

With this knowledge, the follower can calculate a value for W that it will indirectly use for deciding when to request the leadership. In every packet sent by the sender there is information about the current server sending rate in the `rate` field ($W_{\text{sender}}/\widehat{rtt}_{\text{sender}}$). Followers use this information to calculate a moving average of the difference between their expected rate ($W_{\text{follower}}/\widehat{rtt}_{\text{follower}}$), and the rate announced by the sender, where W_{sender} is the window size calculated by the sender, $\widehat{rtt}_{\text{sender}}$ is its RTT estimation and W_{follower} and $\widehat{rtt}_{\text{follower}}$ are

analogous values calculated by the follower host. A *Challenge* packet is generated by the followers every time this difference increases and it is larger than the equivalent of increasing W_{follower} with N_{extra} packets; that is, if

$$\frac{W_{\text{sender}}}{\widehat{rtt}_{\text{sender}}} > \frac{W_{\text{follower}} + N_{\text{extra}}}{\widehat{rtt}_{\text{follower}}}. \tag{6}$$

The N_{extra} value is a tradeoff between having a transmission that is completely fair in all the network links with competing flows, but with a high frequency of leader changes, and one that may take more bandwidth in some links, but is much more stable. In any case, the amount of extra bandwidth allocated is bounded by N_{extra}, so VLMCC can be adjusted to be as much fair as needed.

The *Challenge* packet is filled with information about the current $\widehat{rtt}$ (`rtt`) as measured by the receiver, the calculated rate (`rate`) and the follower's window size (`win_size`). If the sender accepts the challenge, it will use the follower's window size immediately as the new window size, and it will assume all packets sent before the new leader election happened have been acknowledged, even if this is not the case. It would make no sense to react to losses in those packets, when the sending rate, RTT and leader have changed.

In any case, it will be up to the sender to either accept the challenge or ignore it. For example, the sender can decide to ignore a challenge if a new leader has been elected in the last RTTs to let the situation stabilize before revoking that new leader. If the new sending rate would be too low, the sender can also decide not to attend the challenger demands. The challenger can resolve to abandon the transmission if it is not able to cope with the minimum required rate.

2.4 Leadership Loss

As stated in Section 2.1, VLMCC needs the presence of a host responsible of acknowledging every packet sent by the sender. If this leader host is missing, the sender must manage somehow to encounter a substitute. In case the protocol did not provide a mechanism for selecting a new leader, the congestion window size at the sender would drop to 1, and packets would only be sent due to timeouts. Moreover, as probably every host in the transmission would be able to admit higher throughput, no challenges would be produced, and the no-leader situation would perpetuate itself. To avoid this deadlock, the protocol needs to solve two problems, the first is detecting a leadership loss, and the second is being able to recover from it without causing *feedback-implosion*.

There is obviously no reliable way to know that the leader host is down or that it has dropped its connection, because in the first case it may not be able to warn the sender. For this reason VLMCC uses a heuristic to detect leadership loss. The sender assumes that the leader is not responsible when more than a whole window of packets is sent without hearing a single acknowledgment.

Once the leader loss has been detected the sender has to elect a new leader. For this, it ignores the fact that it is not getting acknowledgments and keeps on sending a whole window of packets during each RTT, so that the connection throughput does not degrade unnecessarily. As it is a fact that all the nodes were

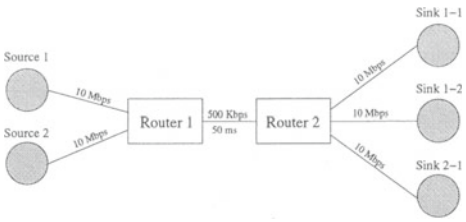


Fig. 3. Simulation topology (delays for links for which no delay is specified is just 0. Higher delays could be used, but we were interested in the bottleneck sharing behavior). All the routers use simple FIFO queues.

not experiencing worse network conditions than the leader, the sender increases the window size in one packet after each RTT to provoke a *challenge* from some host in the network. Once this *challenge* arises, the sender can safely choose the challenger host as the new leader.

3 Experimental Results

We have made several tests to our implementation of the VLMCC protocol with the help of the *ns-2* [14] network simulator. Throughout this section we will discuss the most illustrative experiments that we have performed. We will try to show tests for validating most of the desired properties listed in Section 1.

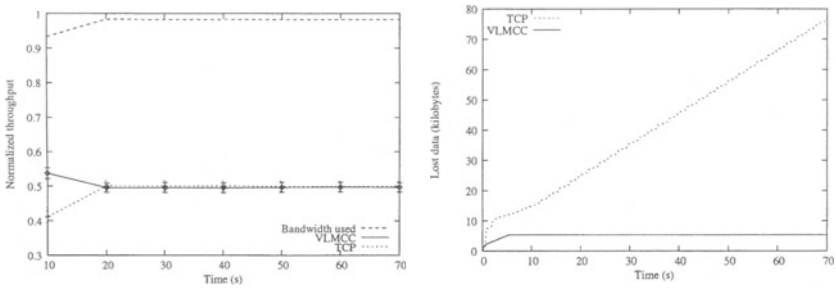
We have represented in Fig. 3 the topology used in the first simulation. We will use this topology to test some basic fairness properties of the protocol. The network shown consists of five end node hosts engaged in two different transmissions. One from *Source 1* to both *Sink 1-1* and *Sink 1-2* and the other one from *Source 2* to *Sink 2-1*. We will be interested in observing how the connections share the bandwidth of the bottleneck between both routers.

The first simulation presented here tries to examine the inter-protocol fairness properties of VLMCC; in particular we are interested in the fairness degree of VLMCC when confronted with TCP flows, as those represent the predominant traffic in the current Internet. To this end, we have run a simulation using *Source 1* as the VLMCC source and *Sink 1-1* and *Sink 1-2* as its receivers, and *Source 2* as the sender of an TCP flow with destination *Sink 2-1*.[3]

Fig. 4(a) shows the obtained results. The simulation has been repeated several times changing the relative start time of both flows, and the averaged throughput has been represented. The VLMCC session manages to equitably share the bottleneck bandwidth with the TCP flow. Only at session startup, when both flows have not still reached their steady state, there are little differences in the obtained bandwidth.

In Fig. 4(b) we show other interesting data obtained in the previous simulation. Instead of representing the received data, we depict the amount of data lost

[3] Unless otherwise noticed, the packet size for all simulations is 1 000 bytes and the TCP version employed is TCP Reno.



(a) Throughput obtained. Error bars for a 95% confidence interval.

(b) Accumulated data lost.

Fig. 4. One VLMCC connection and a TCP flow sharing a single bottleneck.

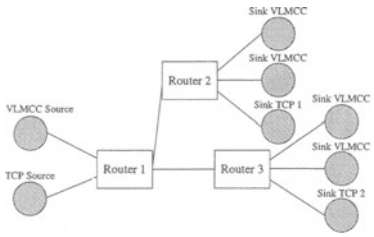


Fig. 5. Simulation topology used to test the stability of VLMCC. All the links have a bandwidth of 10 Mbps and 1 ms delay, except the links between the routers, that have 1 Mbps bandwidth. All the routers use simple FIFO queues.

by each flow. Because TCP uses packet losses as the means to detect congestion and adjust its sending rate, it suffers packet losses all along the simulation; meanwhile, VLMCC manages to find an appropriate sending rate after some seconds of operation, and does not suffer from any packet loss from that instant. Because VLMCC is designed with multimedia streaming requirements in mind, the fact that it suffers from less packet losses is greatly important, as that is directly related to the perceived transmission quality. In real-time streaming, packet losses can not usually be recovered, so it is important to avoid them before they happen. The pro-active window management algorithm employed by VLMCC manages to accomplish this goal fairly well, as shown.

The following simulation tests the ability of VLMCC to adapt quickly to changes in the available network resources. Fig. 5 shows the topology used in the test. During the simulation a VLMCC session is run originating from the node VLMCC *Source* and having all VLMCC *Sink* as destinations. A TCP connection is established between hosts TCP *Source* and *Sink* TCP *1* in the time intervals $[10, 25]$ and $[41, 60]$. Finally, another TCP connection is established between nodes TCP *Source* and *Sink* TCP *2* from second 27 until second 45. The

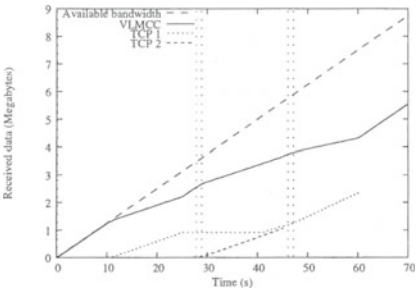


Fig. 6. Accumulated data for both a VLMCC connection and two TCP sessions.

purpose of this simulation is to test if, as the links between the routers become congested, the VLMCC session leader changes and VLMCC is able to adapt and share the bandwidth fairly in both links.

The results are shown in Fig. 6. The vertical lines represent the instants when a new leader is elected in the VLMCC session. The other curves show the amount of properly received data by each connection. The slope of these curves represents the obtained throughput. At the beginning of the simulation one of the VLMCC receivers connected to *Router 2* is selected as the leader, so, when the first TCP flow starts at second 10 no leader change is needed, as both the VLMCC leader and the TCP flow share the same bottleneck. In the graph it can be noticed how the throughput of the VLMCC session decays. At second 25 when the first TCP flow stops, the VLMCC throughput increases again, until second 27 when the second TCP flow starts both VLMCC receivers under *Router 3* try to become the leader, to be able to adapt the throughput to the bandwidth in the new bottleneck. When, at second 41, the first TCP flow restarts, VLMCC changes nothing, as both bottlenecks are in the same conditions, but, when the second TCP flow ends (second 45), one of the receivers under *Router 2* becomes the leader again to be able to share the bandwidth in the link *Router 1-Router 2* in a fair manner. As seen, VLMCC reacts quickly in dynamic scenarios moving the leader location to a host in the congested part of the network.

In the last simulation we want to show the behavior of the protocol in a realistic scenario (similar to one used in [15]), such as the one depicted in Fig. 7. Here, a VLMCC session is established between one long-lived VLMCC source and three different sinks. At the same time a TCP connection, some UDP traffic and some HTTP sessions populate the network. Fig. 8(a) shows that even in this complicated situation VLMCC is capable of behaving in a TCP-fair manner against the competing TCP flow. We have chosen to represent the accumulated data received in order to see more clearly the results in the long range, given that the instantaneous throughput is just too noisy.

In Fig. 8(b) we plotted the congestion window size of both the TCP flow and the VLMCC sender. Although both experience noticeable variations due to the congested state of the network, the oscillations of the VLMCC window are

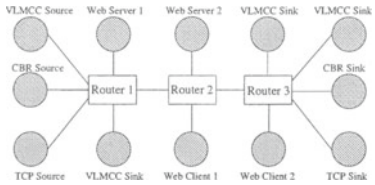
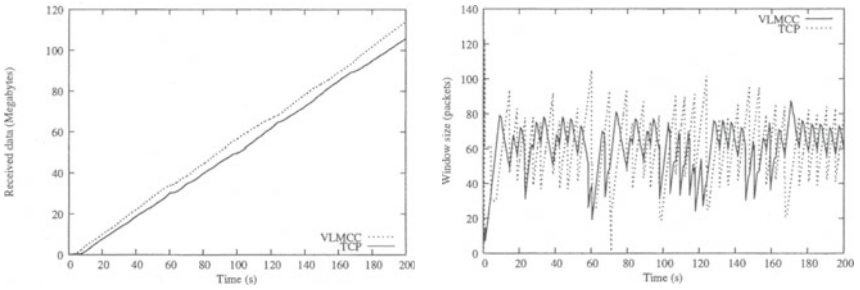


Fig. 7. Simulation topology for testing the VLMCC performance in a complex scenario. All the links have a bandwidth of 100 Mbps and 24 ms delay, except those that interconnect the routers, that have a bandwidth of 10 Mbps and 1 ms delay. All the routers use simple FIFO queues.



(a) Accumulated data received. (b) Congestion window size.

Fig. 8. Bandwidth sharing between VLMCC and TCP sessions.

less severe than those of TCP. Although in isolation VLMCC would probably be able to reach a constant window size, the fact that the network is congested makes VLMCC suffer from packet losses and so its behavior is closer to that of TCP-Reno.

4 Conclusions

We have presented a multicast congestion control protocol suitable for the streaming of real-time multimedia data. The main innovations of the protocol are the use of the variations in the round trip time to detect congestion at the receivers, and the decentralization of the leader election process.

Because of the use of variations of the RTT to detect congestion VLMCC is able to react quickly to congestion. This is important, because slow reaction has been shown to loose throughput over faster ones. This is one of the main drawbacks of equation-based protocols [10].

The decentralization of the leader election process greatly reduces the amount of feedback that receivers send back to the sender, up to the point that there is no need to regulate it to avoid feedback-implosion. Unlike other schemata where

the sender is in charge of selecting the leader, VLMCC receivers can detect when they need to become the leader, reducing the load suffered by the sender.

Simulation results show that the protocol behaves in a TCP-fair manner and thus it should be safe for deploying in the Internet. Finally all the configuration parameters of the protocol are time-independent, what makes unnecessary reconfiguring it for different link bandwidths or delays present in the network.

References

1. Tan, W., Zakhor, A.: Real-time internet video using error resilient scalable compression and TCP-friendly transport protocol. IEEE Transactions on Multimedia **1** (1999) 172–186
2. Widmer, B., Handley, M.: Extending equation-based congestion control to multicast applications. In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, ACM Press (2001) 275–285
3. Rejaie, R., Handley, M., Estrin, D.: RAP: An end-to-end rate-based congestion mechanism for realtime streams in the internet. In: Proceedings of the IEEE INFOCOM. Volume 3. (1999) 1337–1345
4. Feamster, N., Bansal, D., Balakrishnan, H.: On the interactions between layered quality adaptation and congestion control for streaming video. In: 11th International Packet Video Workshop. (2001)
5. Yang, Y.R., Lam, S.S.: Internet multicast congestion control: A survey. In: Proceedings of ICT 2000, Acapulco, Mexico, ICT 2000 (2000)
6. Jacobson, V.: Congestion avoidance and control. In: Symposium proceedings on Communications architectures and protocols, ACM Press (1988) 314–329
7. Floyd, S., Fall, K.: Promoting the use of end-to-end congestion control in the Internet. IEEE/ACM Transactions on Networking (TON) **7** (1999) 458–472
8. McCanne, S., Jacobson, V., Vetterli, M.: Receiver-driven layered multicast. In: ACM SIGCOMM. Volume 26,4., New York, ACM Press (1996) 117–130
9. Rizzo, L.: pgmcc: a TCP-friendly single-rate multicast congestion control scheme. In: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, ACM Press (2000) 17–28
10. Bansal, D., Balakrishnan, H., Floyd, S., Shenker, S.: Dynamic behavior of slowly-responsive congestion control algorithms. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM Press (2001) 263–274
11. Rhee, I., Balaguru, N., Rouskas, G.N.: MTCP: Scalable TCP-like congestion control for reliable multicast. Computer Networks: The International Journal of Computer and Telecommunications Networking **38** (2002) 553–575
12. DeLucia, D., Obraczka, K.: Multicast feedback suppression using representatives. In: INFOCOM (2). (1997) 463–470
13. Brakmo, L.S., O'Malley, S.W., Peterson, L.L.: TCP Vegas: New techniques for congestion detection and avoidance. ACM SIGCOMM Computer Communication Review **24** (1994) 24–35
14. NS: ns Network Simulator (2003) http://www.isi.edu/nsman/ns/.
15. Bansal, D., Balakrishnan, H.: TCP-friendly congestion control for real-time streaming applications. MIT Technical Report, MIT-LCS-TR-806 (2000)