# pgmcc: a TCP-friendly single-rate multicast congestion control scheme*

Luigi Rizzo
Dip.Ing.Informazione, Univ. di Pisa
http://www.iet.unipi.it/ luigi/
luigi@iet.unipi.it

## ABSTRACT
We present a single rate multicast congestion control scheme (pgmcc) which is TCP-friendly and achieves scalability, stability and fast response to variations in network conditions. pgmcc is suitable for both non-reliable and reliable data transfers; it uses a window-based TCP-like controller based on positive ACKs and run between the sender and a group's representative, the *acker*. The innovative part of pgmcc is a fast and low-overhead procedure to select (and track changes of) the acker, which permits us to consider the acker as a *moving* receiver rather than a *changing* one. As such, the scheme is robust to measurement errors, and supports fast response to changes in the receiver set and/or network conditions. The scheme has been implemented in the PGM protocol, and the paper presents a number of experimental results on its performance.

## Keywords
Congestion control, multicast, TCP, fairness.

## 1. INTRODUCTION
This paper presents a single-rate multicast congestion control scheme which is suitable for single-source multicast protocols, with or without support from network elements.

Congestion control schemes require some form of feedback, from receivers or network elements, to adapt the source's data rate to the network capacity. In many multicast protocols, some feedback is already available, to provide either data integrity (as it is the case for reliable multicast protocols) or gather statistics on receivers (e.g. RTCP reports), so one would like to make use of the existing feedback information for congestion control purposes as well.

Unfortunately, it is typical for multicast protocols to use techniques such NAK suppression [14] or FEC-based repairs [13, 20], which give substantial advantages from the point of view of scalability, but result in delayed feedback. Such delays, when introduced in the control loop, would make the system very unresponsive to variations in the network conditions. Stability issues aside, a lack of responsivity is also a potential source of unfairness: when competing with other traffic which reacts to congestion with shorter time constants, the latter might be driven to a very low throughput by the more aggressive, slow responsive flows.

The scheme proposed in this paper tries to achieve fast response, while retaining the scalability of existing feedback schemes, by electing a group representative (*acker*), and running a tight control loop between the sender and the acker. The latter is chosen as the receiver with the worst throughput (according to the scheme used in the control loop) among group members.

The main difficulty of this approach is that the "worst" receiver might change rapidly, and handling the switchover between different ackers might be slow and extremely difficult as experienced by other authors [2, 6].

The key and innovative idea of our scheme is to consider such switches not as a *change* in the acker, but rather as a *move* of the acker to a different location. By acting this way, we can assimilate some of the effects of switching acker to network perturbations (such the presence of path changes, multipath, or changes in overall load), which we already have to deal with in the unicast case.

Our scheme deals effectively with acker changes, and responds quickly to variations in network conditions. It also does so in a way that is amenable for incremental deployment, as it can work without any assistance from network elements yet make good use of it if available. All the components that we include in our scheme are carefully chosen to be scalable, both in terms of state, traffic and processing overhead. This sometimes limits performance (e.g. we might achieve faster response by requesting positive ACKs from all receivers), but also permits us to keep the protocol simple, robust and scalable.

In the following we will concentrate on the discussion of our single-rate multicast congestion control scheme. The discussion of the limitations of single-rate multicast protocols (compared, e.g., to multi-rate multicast protocols) is beyond the scope of this paper.

*This work has been partially supported by Cisco Systems, and the Ministero della Pubblica Istruzione.

*SIGCOMM* 2000, Stockolm, Sweden.

The paper is structured as follows. In Section 2 we present the problem of multicast congestion control and give a short overview of related work, also mentioning multi-rate schemes. In Section 3 we present our scheme in detail, describing the various components, motivating the design choices, and discussing its limitations. Experimental results are presented in Section 4, and we draw some conclusions and present future work in Section 5.

## 2. RELATED WORK

The problem we are considering in this paper is the regulation of the transmit rate of a single-source multicast session, in such a way that the source's data rate adapts to variations in network conditions and receiver population. We do not make particular assumptions on the group size or the heterogeneity of the links leading to the receivers, nor on the overall reliability of the data delivery component of the protocol. We do expect that receivers are able to send loss reports towards the source, either directly or by means of a intermediaries such as routers or other receivers.

The problem of multicast congestion control has received some attention in the literature. The two dominant approaches are based on single-rate and multi-rate schemes.

### 2.1 Single-rate schemes

In single-rate schemes, all receivers get the same data rate, and the source adapts to the slowest receiver. These schemes are nice in that they do not require the source to transmit multiple streams or use special data coding. Furthermore, many single-rate (reliable) multicast protocols have been proposed which try to implement a TCP-like service over multicast, so there is some interest in adding congestion control to such protocols [4, 5] especially when deployment in the Internet is desired.

Single-rate schemes have known limitations in presence of large or heterogeneous groups: a single slow receiver can drag down the data rate for the whole group. Furthermore, uncorrelated losses at receivers are not easy to handle, and an improper aggregation of feedback is likely to cause the so called "drop-to-zero" problem [23], where the sender's estimate of the loss rate is much higher than the actual loss rate experienced at every single receiver.

Some researchers have proposed rate-based schemes where the sender uses loss reports to update the transmit rate [10]. These schemes usually work on coarse timescales (1 second or more), and are in a sense simpler to adapt to multicast than window-based schemes as they do not need per-packet feedback. The rate is often computed according to some formula such as the TCP equilibrium equation [8, 15], which relates the throughput to the loss and RTT (these schemes are sometimes called "equation-based" for this reason).

TFMCC [4] presents some basic algorithms that can be used within multicast rate controllers, especially with NACK and Hierarchical ACK based protocols. Some of the mechanisms and ideas described in TFMCC have been used in the design of `pgmcc`.

Golestani [3] has studied window versus rate-based schemes and methods on how to extend them to multicast while preserving TCP fairness. The use of a window-based scheme for multicast congestion control has been proposed by Rhee [16], whose scheme (MTCP) requires a tree-structure for aggregation of feedback from the receivers. One of the main differences between `pgmcc` and MTCP is that we do not require per-packet feedback from all receivers, so `pgmcc` scales much better in absence of aggregation nodes (sender agents in MTCP terminology).

Other researchers have proposed the use of representatives for congestion control purposes [2, 6], discussing mechanisms for the selection of representatives. A congestion control scheme based on representatives has been suggested in [6], but the authors mention the weakness of their scheme when reacting to change of representatives.

### 2.2 Multi-rate schemes

Multi-rate schemes are based on the ability to generate the same data at different rates over multiple streams (generally organized as cumulative layers), either at the source, or as a result of a filtering/distillation process done by intermediate elements such as routers or transcoders. Receiver try to listen to one or more streams matching their capacity, thus effectively realizing a partitioning of the set of receivers into different groups. This approach is suitable to both audio and video streams, and to reliable data transfer by using proper coding techniques [1, 18].

The advantage of multi-rate schemes is that receivers with different needs can be served at a rate closer to their needs, rather than having to match the speed of the slowest receiver in the group. This flexibility is paid in terms of coding costs, some bandwidth inefficiency, and possibly a more coarse match of source and receiver data rate.

Several schemes have been suggested in the literature to support layered congestion control [9, 22, 7]. All of them are generally based on estimates done by receivers on the available data rate; some of them try to achieve TCP fairness, and include techniques to achieve synchronization among receivers behind the same bottleneck.

## 3. PGMCC

The goal of our scheme is in the first place to make the sender transmit no faster than the TCP-fair rate available at the slowest (as defined later) receiver. Our scheme can be quickly described as follows:

> *The sender continuously monitors receiver reports embedded into NAKs, selecting (and tracking changes of) a group's representative, the* acker*, as the receiver with the worst throughput according to the control scheme being used. A window-based congestion control scheme similar to TCP congestion control is run between the sender and the acker, which will send positive ACKs for each data packet.*

`pgmcc` operates end to end, and requires small constant state and a minimal amount of computation at both sender and receivers. We want to emphasize that, while our scheme involves ACKs and NAKs, we do not make any assumption on the reliability of the data transfer. This makes our scheme applicable equally well to unreliable data transfers.

The window-based control differs from TCP congestion control in some small but important details, such as the use of distinct "windows" for rate and for reliability/flow control; a different retransmission behaviour; the use of sender-based RTT measurements for selecting the representative, and not just for determining timeouts; and a slightly different ack clocking scheme in presence of switchover of representatives. The acker election/tracking mechanism is the main innovation of our scheme and it allows us to handle switches of the acker in a very robust way.

We had three main goals in this work: develop a scheme which is scalable, responsive, and amenable to incremental deployment.

Scalability is achieved by decentralizing functionality and state as much as possible. This is done using solutions (e.g. NAK suppression via randomization; the selection of a *single* node in charge of sending ACKs) which only require a constant amount of state on servers, routers and receivers, independent of the group size.

Fast response is achieved by introducing positive ACKs from the acker, which permit a more timely distribution of information than NAKs.

Incremental deployment is possible because our scheme operates end to end, but can take advantage of hop-by-hop support if present, whether partially or fully available. As an example, our scheme makes use of router-based feedback aggregation, but can work even without that. It does not make use of receiver-based RTT measurements, because this would require either globally synchronized clocks (e.g. using GPS) or a router-based protocol to support the measurements.

## 3.1 PGM

We have developed our scheme in the context of the PGM multicast protocol (hence the name "pgmcc"), although our work is of general applicability and not bound in any means to PGM. Thus we start this section with a very short description of the subset of PGM of our interest.

PGM [21] is a single-sender, multiple-receiver multicast protocol which gives improved reliability over the basic IP multicast by making use of NAK-based retransmission requests. The usual feedback suppression techniques based on randomized delays are used in PGM to achieve scalability. Furthermore, PGM can make use of (but does not depend on) router support for feedback aggregation and selective repair forwarding. Specifically, PGM-enabled routers will do hop-by-hop NAK forwarding, suppressing replicated NAKs coming from the same subtree. They will also do selective forwarding of repair traffic only to those branches from which a matching NAK was heard.

In order to make feedback go through the same path as the (forward) data traffic, PGM uses special control messages called Source Path Messages (SPM), which are rewritten hop-by-hop by PGM-enabled routers, and inform nodes on the identity of the next PGM node upstream.

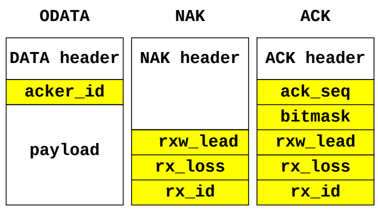


**Figure 1: Packet formats. The grey areas correspond to PGM options inserted to support congestion control.**

Because the PGM specification does not include a congestion control scheme, PGM sources typically transmit at a pre-set data rate. With the introduction of our control scheme, the PGM rate limiter only serves to limit the maximum data rate of the session.

Our modifications to PGM involve the addition of some options to PGM packets (see Figure 1), one new packet type, and some modifications to the sender and receiver procedures. There is a high level of compatibility between regular PGM and `pgmcc` senders and receivers. A `pgmcc` receiver can talk to standard PGM senders with no modifications; a standard PGM receiver can listen to a `pgmcc` session provided that there is one `pgmcc` receiver acting as the acker (otherwise the session will incur a stall on each packet for lack of ACKs); and, in our implementation we can dynamically disable the congestion control procedures on the sender to make it act as a regular PGM sender[1].

## 3.2 Receiver reports – RTT and loss measurements

Receiver reports are a fundamental component of our scheme. They are sent back to the sender as NAK options (see Figure 1), and are made of three fields:

- the identity of the receiver, `rx_id`;
- the highest known sequence number, `rxw_lead`
- the loss rate measured locally, `rx_loss`.

As we will see in Section 3.5, the latter two fields can be used by the sender to estimate the throughput of the receiver, and select the slowest one as the representative of the group. The estimate relies on a measurement of both the Round Trip Time (RTT) and the loss rate for each receiver.

### 3.2.1 *RTT Measurement*

A classical way to measure the RTT without synchronized clocks is to include a timestamp in each packet from the

[1]Only trivial modifications are needed to make a sender dynamically recognise the absence of `pgmcc` receivers and switch to a standard PGM, mode where it works in absence of ACKs.

source, and let receivers echo back the most recently received timestamp. The echoed timestamp should be corrected with the difference between the time of reception and the time the feedback is actually sent, so that delays in sending the feedback (e.g. those used for NAK suppression) are not erroneously interpreted as part of the RTT. This method can give reasonably precise estimates of the RTT, but has two drawbacks: it requires additional information in each packet from the source, and also depends on the resolution of the clock at each receiver. If the latter is too coarse, the correction factor might introduce a large variance on the RTT estimates, biasing the results of the measurements in favour or against some receivers. Because we expect to deal with a large population of heterogeneous receivers, we cannot depend on the availability of a high resolution clock at all receivers.

To avoid these problems, in `pgmcc` we chose to measure the RTT in terms of packets: the sender simply computes the difference between the most recent sequence number sent and the `rxw_lead` value coming from the receiver. To do this we do not need to send timestamps or rely on the timer resolution at the receiver; on the other hand, for a path with a given RTT (measured in seconds), the value in packets computed by `pgmcc` will vary depending on the actual data rate. However this variation applies in the same way to all receivers, so it is not a source of discrimination among receivers. Furthermore, the RTT measurement in `pgmcc` is only used for comparing receivers, not for the actual selection of transmit rate, so any discrepancy between the real and the measured RTT cannot influence the inter-protocol fairness. NS simulation using both methods (with precise clocks at the receivers) showed that the use of time-based RTT measurements does not yield any better behaviour in any of the configurations tested.

### 3.2.2 *Loss measurement*

To measure the loss rate, each receiver interprets the packet arrival pattern as a discrete signal (1 for lost packets, 0 otherwise) and passes it through a discrete-time linear filter (Fig. 2), whose response (and computational costs) can be chosen as appropriate. In our case we used a first-order lowpass filter, whose equation is

$$Y_i = WY_{i-1} + (1-W)x_i$$

(computations are done in fixed point arithmetic with 16 fractional bits, which are quickly implemented using basic integer arithmetic operations and shifts). We chose (rather empirically) a value for the constant $W = 65000/65536$, which corresponds to a corner frequency of approximately 0.0013 packets$^{-1}$, and from our simulations gives reasonable smoothing of the input signals at the loss rates (below 10%) of interest. The actual value of $W$ is not terribly critical for the operation of the protocol.

It is noticeable that in our computations we do not make use of absolute times, but only of packet sequence numbers. This means that the properties of our system (including responsivity) can be evaluated independently of the actual data rates at which the scheme is being used.

An example of the output of the filter is shown in Figure 2, for three different values of the constant $W$ (the one we

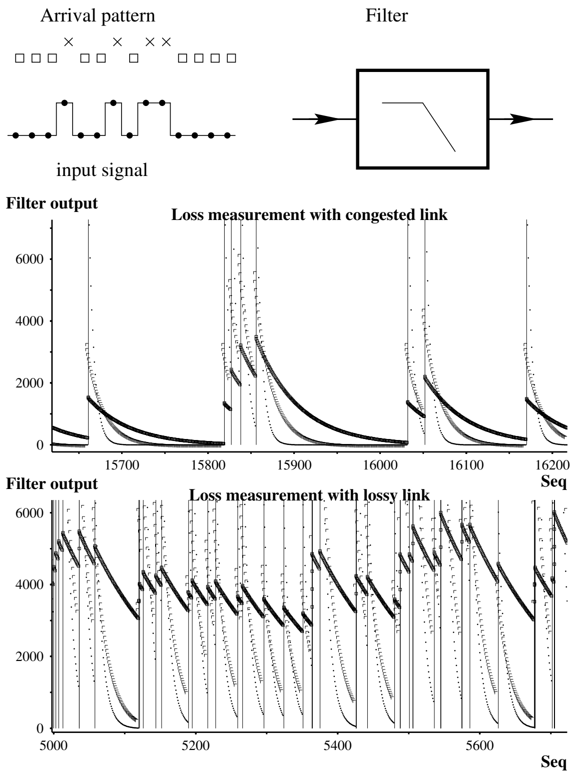


**Figure 2: Loss rate computation at receivers. Above: principle of operation. Below: two examples with non-lossy and lossy links, and three different values for $W$. The $y$ axis is the output of the filter multiplied by $2^{16}$**

used correspond to the thick square marker), and two different loss patterns. The $y$ axis is the output of the filter multiplied by $2^{16}$, whereas the vertical bars correspond to actual losses. The top graph presents a sample of the loss behaviour of a 60Kbit/s link with a single session – losses are rather sparse and the overall loss rate is low. The bottom graph instead represent a link with 5% random loss, modeling the opposite case of an overloaded link with very high statistical multiplexing.

In both cases, we note that loss measurements (the outputs of the filter) are affected by relatively large deviations, unless one takes averages on a very high number of samples. However by doing this, our estimates would become too insensitive to short term variations. We will discuss the effects of these uncertainties in Section 3.5. We also plan to investigate, as future work, the techniques used in TFRC [12] for measuring losses.

## 3.3 Acknowledgements

For each data packet (but not for retransmissions), one of the receivers is in charge of sending positive acknowledgements (ACKs). The identity of the acker is carried as a PGM option in each data packet (Figure 1).

ACKs (see Figure 1) contain the same loss report as NAKs, and a couple of additional fields, namely the sequence number (`ack_seq`) of the data packet which elicited this ACK, and a 32-bit bitmap (`bitmask`) indicating the receive status of the most recent 32 packets. This scheme serves to transmit each ACK multiple times, thus allowing the sender to recover lost ACKs, and deal properly with out-of-order ACK delivery.

Compared to TCP, the format and handling of acknowledgements is peculiar for two reasons. First, in our scheme (and in PGM) repair packets can be transmitted with a significant delay from the loss detection. Thus we cannot implement a cumulative acknowledgement mechanism a-la TCP, and we need an alternate scheme to take care of loss of ACKs and out-of-order delivery of data and ACK packets. Secondly, we want to support quick switch of ackers, and this again might cause phenomena which are very similar to multipath effects in unicast communication, such as sudden changes in RTT, and out of order ACK delivery. The presence of the bitmap gives us a better chance to deal with these effects.

## 3.4 Window-based controller

We use a window-based congestion control scheme which is run between the sender and the acker, and mimics TCP congestion control. Because of the late delivery of repairs and the absence of cumulative acknowledgements, the "window" used for congestion control purposes does not correspond to the "window" used for reliability or flow control.

In our scheme, the sender manages two state variables: a *window* $W$, and a *token count* $T$, both initialized to 1 when the session starts or restarts after a stall (i.e. ACKs stop coming in, and a timeout expires). The window $W$ managed by our controller can be interpreted as an estimate of the number of packets in flight (although the latter could be much lower than $W$ in many cases. The real role of $W$ is to determine how fast the window opens, same as in TCP). Tokens are instead used to regulate the generation of data packets: one token is necessary (and consumed) to transmit one packet, and tokens are regenerated by incoming ACKs.

In detail, $W$ and $T$ are updated as follows:

- on session restart, $W = 1, T = 1$;
- on transmit, $T = T - 1$ (consume one token);
- on ACK, $W = W + 1/W, T = T + 1 + 1/W$;
- on loss detection, $W = W/2$, ignore next $W/2$ acks.

The behaviour on normal ACKs mimics TCP's linear increase – the window expands by one for each round trip time. Similar to TCP, we assume a packet loss when a given packet has not been ACKed in a number of subsequent ACKs (this *dupack threshold* is set to 3 in our tests), and reproduce TCP's multiplicative decrease by halving the window. In order to match the number of outstanding packets to the window count, we need to avoid incrementing the token count for $W/2$ acks[2]. Also, we do not react to further congestion events for the next RTT (this is easily achieved by recording the sequence number of the most recently transmitted packet).

[2]This can be done either at once, or by counting only 1/2 token per ACK for the next $W$ acks, or by other similar strategies.

Since ACKs carry the `rxw_lead` value, on a loss detection we can also realign $W$ to the actual number of packets in flight before halving the value. This serves to avoid that errors accumulate over time, leading to window estimates which are much larger than reality, and thus would result in less aggressive window opening.

TCP does exponential window opening after a stall, up to a threshold which is computed adaptively. To avoid stalls when competing with TCP flows in presence of low buffering in the network, `pgmcc` also does exponential opening of the window up to a small, fixed size (6 packets). This is done to quickly open the window beyond the dupack threshold. Whether to make the slow-start threshold adaptive (and possibly let it grow much higher than our value) is a subject for future investigations. The exponential opening of the window is an extremely aggressive behaviour, and very likely to cause congestion. Especially on session startup, there is relatively low confidence on the fact that the selected acker is really the slowest receiver in the group, so we want to keep a more cautious behaviour than TCP.

## 3.5 Acker Election and Tracking

The procedure to elect and track changes in group representatives is critical to the correct operation of our scheme, and is based on the reports received by the sender. The elected acker will control the throughput of the session through its feedback, so we want to switch to a new one when its maximum throughput (allowed by the control scheme when competing with other flows) will be lower than the one of the current acker.

The approach we use is based on the steady-state characterization of the control scheme. Let $\underline{X_i}$ be the operating parameters (e.g. loss rate, round trip time, etc.) for receiver $i$, and assume that we can (at least approximately) compute the throughput $T(\underline{X_i})$ of a receiver with given parameters. We then select the acker as node $i$ in the set $\{R\}$ of receivers such that

$$i : T(\underline{X_i}) \leq T(\underline{X_j}) \forall j \in \{R\}$$

The exact formula to be used for $T()$ depends on the control scheme in use. TCP congestion control has been studied by several researchers, who have derived various versions of the TCP equilibrium equation [8, 15]. A simplified form which suits our needs[3] is

$$T() \propto \frac{1}{RTT\sqrt{p}} \tag{1}$$

[3]at loss rates roughly above 5%, the simple equation largely overestimates the throughput of the session. Thus, a receiver with high loss might be erroneously elected as the acker instead of another one with lower loss but higher RTT. This can be fixed by using the more precise formula presented in [15]. We plan to conduct further investigations in order to determine the likelihood of a scenario where the use of the full formula leads to a more correct behaviour of the protocol.

and NAKs and ACKs carry sufficient information for the sender to compute $T()$ and let it choose the acker (in particular, we compare the $RTT^2 \cdot p$ values as this is cheaper to compute).

Equation 1 models a window-based controller with linear increase and multiplicative decrease, which is exactly the one we use. Note that different controllers yield different relations, e.g. using multiplicative increase produces a relation of the form $T \propto \frac{1}{RTT \cdot p}$. Also remember that for a given controller, the same throughput could be achieved with different combinations of the input parameters.

The selection process does not require knowledge of the whole population of receivers, or the evaluation of $T()$ for all of them. When we receive a NAK from node $j$, we can decide whether to switch to a new acker from the current one (node $i$) by just comparing $T(\underline{X_i})$ and $T(\underline{X_j})$.

We should remark that the acker selection process is unavoidably approximate. While we know some information (possibly including the current data rate) about the current acker, we might be far away from steady-state behaviour. For new candidates the situation is even worse, as we are likely to know only the information supplied in the most recent report. Also, as noted in Section 3.2, the RTT and $p$ estimates are affected by large uncertainties. This might cause frequent switches which do not correspond to real packet losses. Furthermore, the formula used to elect the acker is approximate and derived under assumptions which might not be valid during the switch. As a consequence, it is essential that we do not interpret a change of acker as to a congestion signal. Rather, we assimilate the selection of a new acker to a *move* of the node in charge of sending ACKs to a path with different features. This is possible because for each data packet there is only one acker, and we have procedures to deal with duplicate, out of order and missing ACKs. Should the new acker experience congestion, we will get a timely notification by making use of the new ACKs.

It is interesting to discuss the effect of acker switches on the protocol. In many cases, the presence of multiple receivers might cause a reduction of the throughput because of the way feedback is aggregated. `pgmcc` differs in this respect, because the loss event leading to an acker switch is not interpreted as a congestion event (unless it is also experienced by the current acker).

As a consequence of this behaviour, an increase in the number of receivers with uncorrelated losses might even result in a modest increase in the throughput of the session, especially if acker switches are very frequent. In order to reduce this effect, and because the parameters used to compute the throughput are subject to relatively large measurement errors, the actual selection process only switches to a new acker $X_i$ if $T(\underline{X_i}) < cT(\underline{X_j}), 0 < c < 1$. The constant $c$ serves to bias the decision in favour of the current acker, and in turn to reduce the number of acker switches when two receivers are a similar throughput. Experiments and simulations showed that values for $c$ between 0.6 and 0.8 reduces the number of acker switches without influencing the accuracy of the acker selection process, and with beneficial effects on the protocol itself in terms of scalability and fairness with competing TCP flows.

To evaluate the effect of the constant $c$, consider Fig. 4. In this experiments $c = 1$ is used; the three receiver behind the same bottleneck see the same loss, but even the small variations in the RTT measurements suffice to cause some acker switches. Using $c = 0.75$ on the same experiment removes all (unnecessary) acker switches shown in the figure. We have experienced a similar effect on all other simulation scenarios.

## 3.6 Session startup

With a window based control mechanism, the session cannot proceed without an acker to reconstruct the supply of tokens. Also, a session might stall when, in presence of high loss, incoming ACKs are not able to regenerate enough tokens. So, both at session startup and after a stall, we need to immediately elect one acker. Since the election is based on NAKs, we mark the first data packet sent, after the session starts or after a couple of stalls in a row, to elicit a fake NAK, resulting in the election of an initial acker to keep the ACK clock going.

## 3.7 The role of network elements

In PGM and other protocols, network elements (NE) can be used to implement some kind of feedback aggregation to improve scalability. In the case of PGM, NAKs coming from receivers are filtered by network elements, so that only the first instance of a NAK for a given data segment is forwarded to the source. Subsequent NAKs are suppressed, at least until the corresponding state in the NE is deleted.

At first sight, this kind of filtering might interfere with the acker election process, in that the receiver report for a node with worse throughput might be suppressed by the router. In practice, this possibility is mitigated by the fact that a node with a worse throughput will eventually send more NAKs, and thus has more chances of not being suppressed. Our NS simulations so far showed that suppression does not pose problems, at least in small scale configuration. However, a detailed investigation of this problem is left for further study.

A possible approach to reduce even further the effect of suppression is the following: NE will store the `rx_loss` value for each NAK they have forwarded (since there is already state for that sequence number, the additional amount of state is very limited). Suppression will not occur if a NAK for the same sequence number carries an `rx_loss` higher than the one forwarded upstream.

Because NAK suppression is only used for performance improvement, not doing suppression will not alter the operation of the protocol. While `rx_loss` alone does not permit the estimate of the throughput, it can still give a good indication of a potential acker. Also, this approach is extremely inexpensive for the NE in terms of additional state or computation.

## 3.8 Regulation of repair packets

The mechanism illustrated so far is only in charge of regulating the transmission of data packets, and it does not affect the pacing of retransmissions (RDATA packets)[4]. These are handled in the same way as TCP – i.e. as soon as a NAK comes in, we sent the RDATA packet only subject to the throughput of the rate limiter.

The regulation of retransmissions is being investigated separately, as it also has other implications, e.g. in increasing the effectiveness of NAK suppression in some pathological cases where the delays in the system cause RDATA to cancel state in network elements right before a NAK from another receiver comes in.

However, provided that the congestion control mechanism works for (original) data packets, the lack of regulation of repairs does not render our congestion control scheme ineffective. In fact, as long as the acker is really the slowest receiver, then the percentage of retransmission is kept low by the controller, and the occasional retransmissions will slow down the data flow on the path, and thus the ack clock.

Uncorrelated losses for nodes not slower than the acker also pose no problem (of course within the limits of scalability of the repair method being used), because of the small percentage of retransmissions. If the number of NAKs from receivers slower than the acker becomes large, then they will likely cause a change of acker and bring us in the situation detailed in the previous paragraph.

The only problematic case is that of NAK storms. These can result by a single receiver joining in the middle of a session and trying to recover the initial data (this can be allowed by a specific PGM option), or by bursty losses. These two cases can be dealt with relatively efficiently within the receivers themselves, e.g. by appropriately pacing the generation of NAKs when a large number of repairs is needed.

### 3.9 Use with unreliable protocols

The reliability component of the protocol might be completely absent in some cases, e.g. when it is not necessary that all data reach all recipients. There are several reasons for doing so, e.g. when the information which is transmitted has a limited lifetime, or there is enough redundancy in the data stream to recover from occasional losses (which are kept low by the corrective actions of the congestion control scheme).

In such cases, `pgmcc` can provide to the source some feedback about the throughput, which the application can then use to control the data being transmitted (e.g. amount of redundancy, quality of encoding in case of audio or video streams, etc.). This can be of particular importance for applications with real-time requirements where the source must adapt its rate to the session's rate to avoid the building of very large queues.

The first kind of feedback is the content of receiver reports, i.e. loss rate and round trip time, contained into NAKs and ACKs. Using these parameters, for example, a source using FEC can set the amount of redundancy to be used for the session, or a real-time application such as a distributed game can tune its timings, etc..

A second kind of feedback comes from the token generation process. As data transmissions are triggered by the generation of tokens, it is possible for the transport protocol to signal the availability of new tokens to the application, so that the latter can generate new data on the fly and exercise a better control on the information which is transmitted, rather than leaving the transport layer the task of selecting which packet to drop in order to keep queues short.

## 4. EXPERIMENTAL RESULTS

We have tested the behaviour of `pgmcc` using both simulations with the NS simulator [11] and experiments with an implementation of the protocol on a real testbed.

Large scale experiments involving up to 200 receivers have been performed with NS. These experiments were used mainly to test the scalability of the protocol, and to analyse the sensitivity to features such as time vs. sequence number-based RTT measurements etc. Also, NS simulations have been performed to validate the results of the experiments which have been run using an actual implementation of the protocol on a real network.

Small scale experiments were performed on an experimental testbed made of a number of hosts running an actual implementation of `pgmcc`, and communicating through some bottleneck links. The bottleneck links were emulated using dummynet[17], a flexible link emulator which allowed us to run the experiment in a tightly controlled and reproducible testbed. During experiments, we have distributed senders and receivers (both for `pgmcc` and for TCP) on multiple workstations, and run the tests on a number of different topologies with various configurations of the bottleneck link(s).

For the experiments discussed in this paper, two main configurations have been used for the bottleneck links. In the first one, which we call *non-lossy*, the link has fixed capacity, a small propagation delay and FIFO queue (unless otherwise specified, we used 500 Kbit/s bandwidth, 50 ms delay, 30 queue slots). Packet drops in this case are only due to congestion. Because of the relatively large queues, the queueing delay can become as large as 500 ms one way, thus being the dominant component of the RTT.

The second configuration (*lossy*) has a higher bandwidth, a higher delay and some amount of randomly generated losses, again with FIFO queueing (unless otherwise specified, we used 2 Mbit/s, 230 ms delay, 30 KBytes queue, 3% packet loss). This second configuration tries to emulate a link with a high degree of statistical multiplexing. Here the bandwidth available to the flow is determined by the controller's response to the loss and RTT, and is much lower than the link's capacity. Because of this reason, congestion-related packet drops rarely (if ever) occur in this configuration, and the RTT is dominated by the propagation delay component.

Unless otherwise specified, the `pgmcc` flows use a payload of 1400 bytes, whereas TCP packets have a payload of 1460 bytes (`pgmcc` headers and options are slightly larger than TCP headers, so the two packets have approximately the

[4]Note that some protocols might not even use repairs (see Section 3.9).

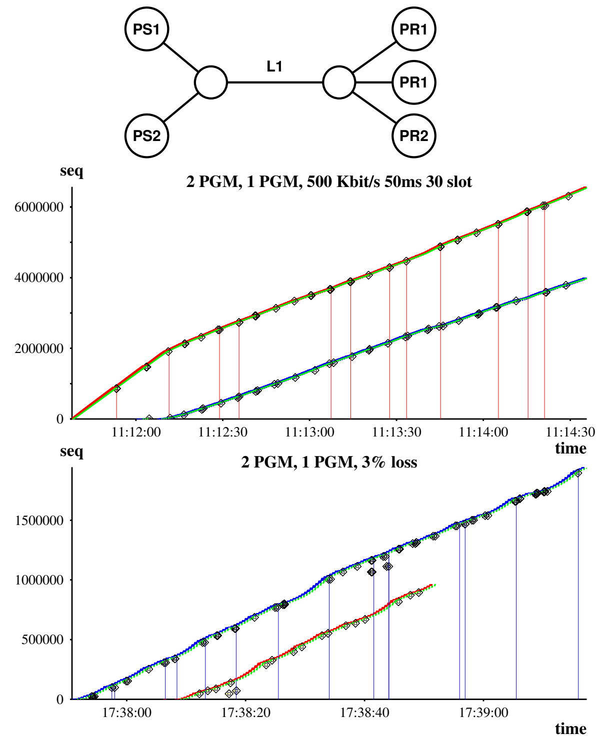


**Figure 3: Intra-protocol fairness, with non-lossy and lossy links. Two pgmcc receivers on one session (started first), one on another session, sharing the same bottleneck.**

same size).

Some initial experiments, which we will not discuss in this paper, have also been run with topologies presenting multiple paths between sender and receiver. This was done to verify the robustness of the scheme to out-of-order data or ACK delivery. Finally, a few test sessions have also been run over the MBone between sites in different countries. However, these experiments were mostly aimed at verifying the correctness of the implementation in term of packet formats and recovery procedures, as the lack of knowledge about the path and the presence, over the MBone, of mostly non-congestion-controlled traffic did not give us any chance of performing significant performance evaluations.

## 4.1 Interpreting graphs

In the rest of this section we will show packet traces logged on the sender side of the bottleneck(s). Above each trace we show the topology used for the experiment; TCP sender and receivers are labeled TS and TR, whereas the PGM senders and receivers are labeled PS* and PR*, the index being the session they belong to.

For both `pgmcc` and TCP sessions, the traces show data and ACK packets (though they can be clearly distinguished only on Figure 5 where we zoom some parts of the trace). In the case of PGM, we also show NAKs, represented by diamonds in the plots. Vertical bars correspond to change of the acker for the PGM sessions. Sequence numbers in the graphs are measured in payload bytes both for TCP and PGM (sequence numbers in PGM headers count packets, not bytes), so we can compare the throughput of the various flows by looking at the slope of the curves. In the experiments shown in Fig. 3 and 4 we have used $c = 1$ to control the switch threshold (see end of Section 3.5) – these were small scale experiments and we wanted to show that switches did not harm the behaviour of the protocol. The remaining experiments and simulations have been run with $c = 0.75$.

## 4.2 Intra protocol fairness

Intra-protocol fairness has been studied by running multiple instances of the protocol through the same bottleneck link, using the topology at the top of Figure 3. Multiple sessions have been run through the same bottleneck link, optionally by adding different delays on the paths to the receivers. As expected, the various instances share bandwidth in a fair way, and with roughly inverse proportionality to their RTT. The graphs in Figure 3 show the evolution of two competing PGM sessions through the same bottleneck, with two different configurations: above is the *non-lossy* case, below is the *lossy* case. One of the sessions (the one started first) has two receivers, the other one only has one receiver.

On the non-lossy link, we see the first session reduce its throughput when the second one starts, and then both share the link evenly. The presence of a second receiver on the first session has no other effect than causing some acker switches, but no impact on the throughput (we will discuss the effect of multiple receivers in Sec. 4.5). On the lossy link, the second session does not appreciably change the response of the first one because there is really no congestion, and the throughput is only determined by the loss rate.

The behaviour of the sessions in both cases is pretty much what we expected. The fact that we run multiple instances of the *same* protocol, makes this kind of experiment not very sensitive to the setting of parameters such as the dupack threshold, window opening slope, slow start threshold, and retransmission behaviour. These parameters will significantly influence inter-protocol fairness.

## 4.3 Inter protocol fairness

A much more critical set of experiment involves competition with other protocols. In this case, even small deviations in the behaviour of the two protocols might cause systematic phenomena leading to starvation of one of them, or to a significant amount of unfairness.

We were particularly interested in a good behaviour of our scheme in presence of competing TCP flows. While we mimic TCP congestion control, there are minor differences among the two schemes. On startup, TCP's exponential window opening is slightly different from the one used in `pgmcc`; there are no delayed ACKs in `pgmcc`; and, after a loss detection, TCP can do an immediate retransmission, whereas `pgmcc` cannot retransmit until a NAK arrives and sets state in routers.

In order to verify the behaviour of competing TCP and `pgmcc` flows, we have run a large number of experiments with the two types of flows and different bottleneck configurations in terms of rate and queue size, both for lossy and

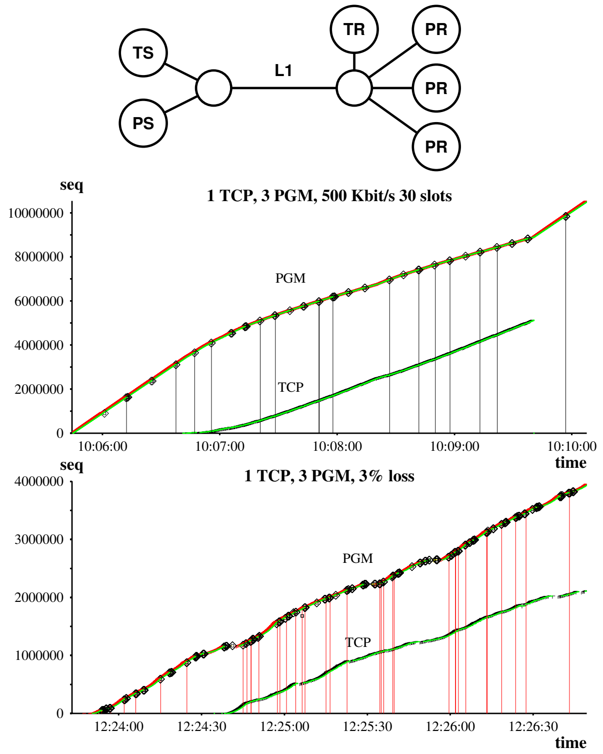


**Figure 4: Inter-protocol fairness. One PGM session (3 receivers) sharing a bottleneck link with TCP. Top: non lossy link; bottom: lossy link.**

non-lossy links.

In general, we see that there is a good sharing of bandwidth between TCP and pgmcc flows in all configurations we tested, and the flows do not starve each other. These results hold independently of the starting order of the flows, and are quite reproducible. However, we should emphasize that on very short timescales, we observed the same phenomena which are present when TCP flows compete against each other: one of the flow might temporarily get a much larger share of the bandwidth because of the generation of a burst of traffic, or because the other flow temporarily reduces its rate. These phenomena are more likely at low bottleneck bandwidths, when the number of packets in transit is low and the traffic can become quite bursty.

A few traces will let us discuss the behaviour of pgmcc versus TCP in more detail. Figure 4 shows the behaviour of one TCP flow competing with one pgmcc session (with up to 3 receivers). The top graph represents a non-lossy link, whereas the bottom graph shows the behaviour on a lossy link. The pgmcc receivers were started at different times (but before the TCP session).

As in the previous experiments, on the non-lossy link we see the pgmcc flow reducing its rate upon startup of the second (TCP) flow, and then both flows progressing at approximately the same rate. At the end of the graph, when the TCP flow terminates, pgmcc regains its original rate. The presence of multiple receivers on the same subnet has practically no effect on the data transfer: all receivers observe the same traffic, so they experience the same losses and measure the same loss rate. NAKs are generated by different receivers, but they are all equivalent and carry almost exactly the same reports. As in the previous experiments, the presence of multiple receivers causes some switches of acker (shown by the vertical bars), but the data rate is not influenced by such switches.

The lossy bottleneck also produces the expected results, with the two session proceeding at approximately the same rate without noticeable perturbations when the second one starts.

## 4.4 Acker selection

Even though the experiment in Figure 4 exercises the acker selection procedures, that one is a simple case because all receivers are at the same location. More difficult cases involve the presence of receivers on different links with uncorrelated losses and possibly different link capacity or delay.

We have run several experiments and simulations to evaluate the behaviour of the acker election process. This was indeed the first set of experiments we ran to understand the dynamics of the acker selection and identify the best mechanism to respond to change in the slowest receiver. The experiments involved a single sender with multiple receivers connected through independent bottlenecks to the source. The bottleneck links were configured with a variety of parameters involving different bandwidths, propagation delays and random loss rates. It was this initial set of tests which evidenced the need of using *all* parameters of the receivers, and not just the loss rate, for selecting the acker (In an initial version of our scheme, we thought we could just select the acker based on the loss measurement, but this belief was easily disproved by building some pathological situations which caused oscillating behaviour). Also from these experiments it soon became clear that the acker selection process was unavoidably imprecise, and that interpreting changes in the acker as congestion notification was going to cause excessive slowdown in the protocol.

Figure 5 shows an experiment run on a topology involving pgmcc receivers on independent links, and competing with a TCP session. Links L1 and L2 were both configured as non-lossy, with 400 Kbit/s, 20 KBytes queue for L1, and 500 Kbit/s, 30 slot queue (approximately 45 KBytes) for L2. The propagation delay was 50 ms in both cases. In the graphs we show the behaviour with the real pgmcc implementation, one receiver at PR1 and one at PR2. We have obtained identical results (including the number of acker switches) in NS simulations with the same configuration and up to 10 receivers at each of PR1 and PR2, and a constant $c = 0.75$ (see Sec. 3.5).

Receiver PR2 is started first, followed by PR1 and then by the TCP connection. The expected behaviour, which we can observe in the graph, is that the pgmcc session first takes the full 500 Kbit/s, reducing to 400 Kbit/s when PR1 joins, and further down to 220 Kbit/s when the TCP flow starts consuming some of the bandwidth on L2. When the TCP flow terminates, PR2 tries to increase the session rate to 500 Kbit/s, causing excessive congestion on PR1 and a

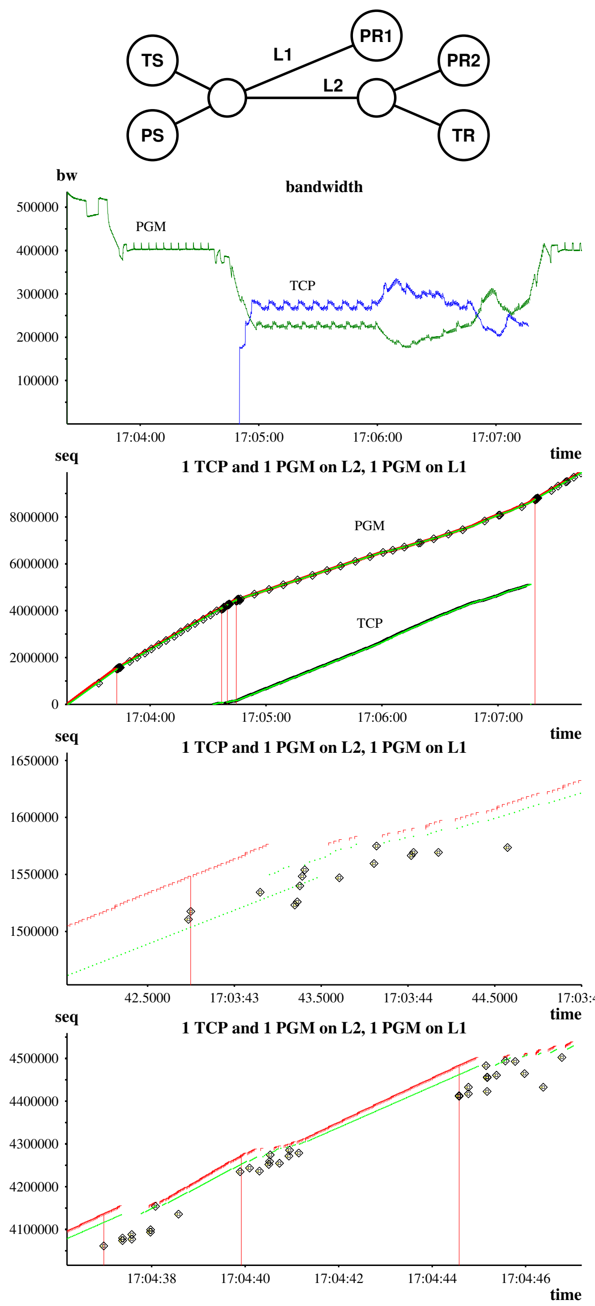


**Figure 5: Acker selection. One TCP session, two pgmcc receivers on different paths with non-lossy links. From top to bottom: topology, bandwidth, time-sequence number plots for the whole session, and close-up of first and second rate change.**

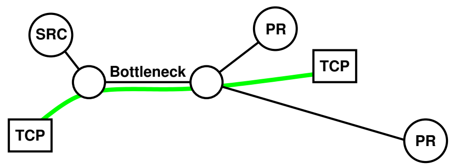


**Figure 6: TCP and PGM session sharing a bottleneck. PGM receivers have different RTTs, some larger, some smaller than the TCP session.**

switch of acker, then settling to L1's bandwidth.

The reader will notice that, when PR1 is controlling the session (between the first and the second acker switch), there are more frequent loss events (represented by diamonds, indicating NAKs). This happens because the queue size, and the RTT which is dominated by the queueing delay, are much shorter for path L1 than for path L2.

It is interesting to have a closer look at the areas near the switches of data rate. The first interesting event is the join of PR1, near time 17:03:42 (roughly corresponding to the first acker switch). Link L1 immediately receives a high-rate stream which fills the buffer and causes a number of losses. On the first two NAKs, which carry a high loss report, PR1 becomes the acker (vertical line). However, there is still a large number of packets queued on L2, which will cause a few more tokens to accumulate and packets to be sent beyond the capacity of L1, thus causing further losses for the next RTT.

When the ACKs from PR1 start coming in, there is a reduction in the RTT (we see the ACKs getting closer to the data packets), which is due to the shorter queueing delay on L1.

Note that right after the first acks from PR1 arrive, the session detects congestion and momentarily stops transmitting. The detection of congestion does *not* occur because the ACKs are out of sequence (as the bitmap contains ACKs for the previous segments as well), but rather because the bitmap contains holes.

In this particular case, reacting to the switchover as to a congestion (i.e. halving window) would prevent a few losses that occur in the next RTT. However this kind of reaction would completely ruin throughput in all other cases described in this Section. We leave for further research the study of mechanism to robustly tell the two different cases and allow this kind of (limited) savings, although we are doubtful on their usefulness.

The effects of the startup of the TCP session is also interesting (Figure 5, bottom). In the session controlled by PR1, there is spare capacity on PR2 and TCP rapidly uses it due to the exponential window opening. This in turn causes a number of losses for PR2 which then becomes the acker, and after a couple of acker switches the two session settle to their steady state behaviour. From this point onwards, the evolution of the session with both flows active is similar to the one of Figure 4.

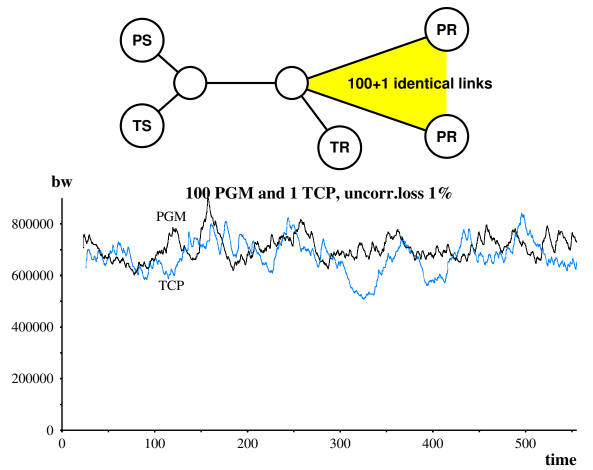


**Figure 7: Effect of multiple receivers with uncorrelated losses. Initially 10 receivers and one TCP, after 300s 90 more receivers join the session, all on different link with 1% random loss.**

To conclude the discussion on acker selection, we want to illustrate the behaviour of pgmcc in the setup of Fig. 6. Here, a TCP session shares a bottleneck with a PGM session with multiple receivers. We assume that the receivers have widely different RTTs, some larger and some smaller than the TCP session, and that all losses are due to congestion and occur on the bottleneck. Depending on the effectiveness of suppression and the spread of RTTs of receivers, pgmcc will select as the acker one of the receivers, but not necessarily the one with the highest RTT. As a matter of fact, in presence of suppression from the network elements, the selected acker will likely be one of the receivers with the shortest RTTs. This behaviour should not be seen as a source of unfairness, but just as a witness of the fact that, in presence of multiple receivers, there is no clear definition of a TCP-fair rate. Multiple TCPs with different RTTs will share bandwidth unevenly, so there is no obvious reason why, in presence of multiple receivers, the multicast session should behave as the slowest rather than the fastest of its members. We can only say that, should the competition with TCP occur on the longest path, the receiver with the highest RTT will experience more losses than the rest of the group and become the acker, thus insuring a correct behaviour on the shared path.

## 4.5 Response to uncorrelated losses

A big problem with single-rate schemes is the behaviour in presence of uncorrelated losses. Depending on how loss reports are handled at the source, the latter might assume an overall loss for the session much higher than the one of each individual receivers. This could cause the so-called "drop-to-zero" problem, i.e. the session's rate dropping to a very low value because of this wrong estimate.

pgmcc tries to avoid this problem by not computing loss rates at the source, but rather just using the estimates done by the receivers, and deferring the response to losses to after the reports from the new acker come in.

We have done some medium-scale evaluations of this behaviour, mostly using NS simulations. Figure 7 gives some indications on how pgmcc works in presence of independent losses with up to 100 receivers. This NS simulation uses one pgmcc source and 100 receivers behind independent lossy links with 1% packet loss rate. An additional link with the same features is used for a TCP flow. At time 0, the TCP session and 10 PGM receivers are started. At time 300, 90 more PGM receivers join the session. As shown from the graph, the presence of the 90 additional receivers does not influence appreciably the throughput of the PGM session.

Much larger scale tests are certainly necessary to investigate this behaviour in more detail. However, such tests cannot be run with simple retransmission-based repairs, or the repair traffic would quickly dominate the actual data traffic on the link from the source.

# 5. CONCLUSIONS AND FUTURE WORK

We have presented pgmcc, a scheme for single rate multicast congestion control which is designed to be scalable, fair with TCP, and achieve fast response by decoupling the reliability/flow control window from the congestion control window. The scheme is suitable for both reliable and unreliable communication protocols, with or without router support.

This scheme has been used to implement congestion control in the PGM protocol, and our preliminary experimental results, part of which are discussed in this paper, show that the system achieves the desired results at least in the configurations we tested. We definitely believe that pgmcc, designed and implemented as described in this paper, can be safely used in the Internet for doing TCP-friendly multicast communication on small/medium scale. A publicly available implementation of pgmcc for BSD-derived systems, including the congestion control part, is available from the author [19].

Throughout the paper we have already mentioned some areas of further investigation, such as using alternate methods to compute the loss rate at receivers; using adaptive rather than a fixed value for the slow-start threshold; replacing the "simple" TCP equation with the more complete model presented in [15]; evaluating the effect of suppression performed by network elements. It will also be important to identify pathological configurations (and their likelyhood in a realistic setup) where pgmcc does not behave as it should. As an example, in configurations where the set of receivers includes low-rtt but lossy links together with high-rtt, congested links, there might be a chance that the wrong acker is chosen and the protocol transmits at a higher rate than it should. While we have not been able to design such a pathological scenario so far, we cannot exclude its existence at this stage.

Changes of the dupack threshold are also under investigation. The main difference here is that TCP does a retransmission right after the detection of enough duplicate acks, whereas pgmcc immediately starts the window halving. Preliminary tests do not show this difference to significantly impact the fairness of the protocols, but we need to gain more confidence on this aspect, especially in situations with few packets in flight.

Another partially open problem, although not only related to congestion control, is the control of retransmission rate. We have not addressed it directly, as this problem only exists for reliable data transfers. However, it is obvious for one that NAK suppression can become quite ineffective if we transmit RDATA immediately upon reception of a NAK. Also, we definitely need some technique to avoid that NAK storms result in bursts of retransmission without any form of regulation. We have suggested in Section 3.8 some of the strategies that we intend to explore in order to deal with these problems.


## Acknowledgements

This work builds on many ideas discussed in the IRTF RMRG meetings. We would like to thank G. Iannaccone, L. Vicisano, J. Crowcroft and M. Handley for their comments on earlier drafts of this work; L. Conti and S. Bertoni for their help in developing a prototype implementation; and V. Benucci for the NS implementation of `pgmcc`.


## 6. REFERENCES


[1] J.W. Byers, M. Luby, M. Mitzenmacher, A. Rege, "A Digital Fountain Approach to Reliable Distributioni of Bulk Data", *ACM SIGCOMM'98, Vancouver, CA, Sep. 1998.*

[2] Dante DeLucia, Katia Obraczka, "Multicast Feedback Suppression Using Representatives", *IEEE Infocom'97*

[3] S.J. Golestani and K. Sabnani, "Fundamental Observations on Multicast Congestion Control in the Internet," *IEEE Infocom'99, Mar. 1999.*

[4] Mark Handley, Sally Floyd, "Strawman Specification for TCP Friendly (Reliable) Multicast Congestion Control (TFMCC)".

[5] A. Mankin, A. Romanow, S.Bradner, V.Paxson, "RFC2357: IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols" `http://www.ietf.org/rfc/rfc2357.txt`

[6] S.K.Kasera et. al, "Scalable Fair Reliable Multicast using Active Services", *IEEE Network, Jan./Feb. 2000.*

[7] X. Li, S. Paul, P. Pancha and M. Ammar, "Layered Video Multicast with Retransmissions (LVMR): Evaluation of Hierarchical Rating Control", *IEEE Infocom'98, San Francisco, CA, Mar.28-Apr.1 1998.*

[8] M.Mathis, J.Semke, J.Mahdavi, T.Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm", *ACM Computer Communication Review, Vol.27 N.3, July 1997.*

[9] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven Layered Multicast", *ACM SIGCOMM'96, August 1996, Stanford, CA.*

[10] Todd Montgomery, "A Loss Tolerant Rate Controller for Reliable Multicast", *NASA IV&V Technical Report NASA-IVV-97-011, Aug. 1997.*

[11] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala, "Improving Simulation for Network Research", *Technical Report 99-702b, University of Southern California, March, 1999. revised September 1999, to appear in IEEE Computer.*

[12] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer, "Equation-Based Congestion Control for Unicast Applications", *ACM SIGCOMM 2000, Stockholm, Aug 2000.*

[13] J. Nonnenmacher, E. W. Biersack, and Don Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission", *IEEE/ACM Transactions on Networking, 6(4):349–361, Aug. 1998.*

[14] J. Nonnenmacher, E.W.Biersack, "Scalable Feedback for Large Groups", *IEEE/ACM Transactions on Networking 7(3):375–386, June 1999*

[15] J. Padhye and V. Firoiu and D. Towsley and J. Kurose, "Modeling TCP throughput: a simple model and its empirical validation", *ACM SIGCOMM'98, Vancouver, CA, Sep. 1998.*

[16] Injong Rhee, Nallathambi Balaguru and Goerge Rouskas, "MTCP: Scalable TCP-like Congestion Control for Reliable Multicast," *IEEE Infocom'99, New York, Mar. 1999.*

[17] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols", *ACM Computer Communication Review, Vol.27, N.1, Jan. 1997*

[18] L. Rizzo, "Effective erasure codes for reliable computer communication protocols", *Computer Communication Review, V.27 N.2, Apr. 1997.*

[19] L. Rizzo, "A PGM Host Implementation for FreeBSD" *http://www.iet.unipi.it/~luigi/pgm.html*

[20] Luigi Rizzo, Lorenzo Vicisano, "RMDP: an FEC-based Reliable Multicast protocol for wireless environments", *ACM Mobile Computing and Communications Review, Vol.2, n.2, April 1998*

[21] Tony Speakman et al., "PGM Reliable Transport Protocol Specification", *Internet Draft, draft-speakman-pgm-spec-04.txt*

[22] L.Vicisano, L.Rizzo, J.Crowcroft, "TCP-like Congestion Control for Layered Multicast Data Transfer", *IEEE Infocom'98, San Francisco, CA, Mar.28-Apr.1 1998.*

[23] B. Whetten, J. Conlan, "A Rate Based Congestion Control Scheme for Reliable Multicast", *Technical White Paper, GlobalCast Communications, Oct. 1998.*