Chirping for Congestion Control - Implementation Feasibility*

Mirja Kühlewind
Institute of Communication Networks and Computer Engineering (IKR)
University of Stuttgart, Germany
mirja.kuehlewind@ikr.uni-stuttgart.de

Bob Briscoe
BT
Sirius House, Adastral Park
Ipswich, IP5 3RE, UK
bob.briscoe@BT.com

ABSTRACT

We present lessons from implementing a bandwidth probing scheme termed chirping as part of congestion control in the Linux kernel. Chirping was first introduced for bandwidth estimation in the pathChirp tool. A chirp is a group of several packets that are sent with decreasing inter-packet gaps in order to continuously probe for spare bandwidth. Current congestion control schemes are either slow to find new bandwidth or they overshoot due to lack of fast feedback information. The attraction of using chirping as a building block for congestion control is to be able to non-intrusively probe for available capacity in a very short time. But implementing chirping is challenging because it requires an exact timing of every packet which is very different to the traditional approach in the network stack. As there are changes needed at the receiver as well, we also discuss a potential approach for deployment. Success in detecting fast feedback information using chirping opens the possibility of future work on new congestion control designs that should be more responsive with less overshoot.

1. INTRODUCTION

A well known problem of Transmission Control Protocol (TCP) congestion control is the inability to quickly reach high throughput in high-speed or highly dynamic bandwidth environments. To overcome this problem, new TCP variants increase their rate more rapidly. But consequently they suffer from overshoot, i.e. they increase too far above the correct rate, leading to significantly greater loss for other flows.

A promising new approach is congestion control based on continually sending so-called *chirps* or *multi-rate probe streams* of packets, as proposed in [9]. A chirp is a group of several packets that are sent with decreasing inter-packet gaps and respectively increasing rate. Thus the packets of the beginning of every chirp are sent at a slower rate, rising to a faster rate at the end. For every chirp the average sending rate is set equal to the rate that the sender wants to achieve. This offers a probing scheme with effective probing rates that vary by orders of magnitude within one chirp, but without overloading the network. Sending with increasing rate within one chirp leads to *self-induced congestion* for a short duration when the per-packet rate is higher than the available capacity. By measuring the relative queuing delays within such a chirp, the sender can estimate the currently available bandwidth, and use this information to adapt the average sending rate of subsequent chirps. Furthermore, this scheme is more robust to noise than other proposed delay-based mechanism.

Rate estimation based on chirping was proposed in [14] for the pathChirp bandwidth estimation tool. The first proposal to use chirping continuously for congestion control was TCP RAPID [9]. With RAPID congestion control the rates of the packets within a chirp are selected in a way that the average rate converges towards the rate estimation of a previous stream. This utilizes the spare capacity on a network link. The algorithm is summarised in Appendix A. However, [9] recognises that implementation in a production operating system will be needed to answer questions that their simulations in ns-2 cannot address.

We have therefore implemented chirping and associated congestion control in the network stack of a production operating system kernel. This paper presents the challenges we encountered and reasoning for the choices we made.

Note that this paper is not an evaluation of the RAPID congestion control algorithm. We certainly implemented it and we make a few remarks about it. But we merely implemented RAPID as an initial placeholder algorithm to exercise our chirping code. Eventually we would like to build our own congestion control algorithm around chirping, but first we have to check whether chirping is feasible. Even if it proves feasible to implement, we then have to check whether chirping can be made sufficiently robust against noise; but that is beyond our present scope.

The goal of this paper is to provide a base on which others can build research in this area. To this end we offer three different types of contribution:

Structured the problem space: We have identified that i) rate adaptation, ii) rate estimation and iii) adapting chirp parameters are three independent sub-problems. A solution to one can be swapped out without affecting the others.

Identified challenges: These can be divided into two sets:

*This work is partly funded by Trilogy, a research project (ICT-216372) supported by the European Community under its Seventh Framework Programme. The views expressed here are those of the author only. This work is also supported by the German Research Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1. Implementation challenges: We cannot use ACK-clocking to determine when to send out each packet within a chirp. With ACK-clocking the arrival of each TCP acknowledgement (ACK) triggers most state transitions, therefore it is a considerable challenge to replace it. Instead each packet release has to be separately timed, which could create a heavy interrupt processing burden. Moreover, chirping does not need the concept of a Congestion Window (CWND) either. However, rather than modify all the mechanisms that use the CWND, e.g. fast retransmit, we track an equivalent of the CWND—a count of the packets allowed in-flight in one RTT.

2. Protocol deployability challenges: For deployability, we would rather only have to modify the sender, treating receiver changes as an optimisation. But we believe the protocol we need for comparing one-way delay measurements cannot work unless new TCP receiver behaviour can be negotiated.

Invented solutions: For example: i) We propose a linear progression to derive the inter-packet gaps that can be implemented with integer arithmetic, in order to limit kernel complexity; ii) We found an easy way to improve the precision of one-way delay measurements without tight timing constraints on sent packets; iii) We discuss the possibility of holding chirp identifiers as soft state in packet headers rather than as hard state at the sender.

The rest of the paper is structured as follows: Section 2 explains chirping. Section 3 describes challenges in the implementation and in protocol and algorithm design and why we addressed them the way we did. Section 4 presents preliminary results and discusses the open issues when using chirping for congestion control. In Section 6 we draw conclusions and outline the further research that will be needed.

2. CHIRPING AS A BUILDING BLOCK FOR CONGESTION CONTROL

A chirp is a logical grouping of N packets. Within a chirp each packet has a higher rate than the previous one. Normally the bits within a packet of size P are sent at line-speed. Therefore, different packet rates are realised by controlling the inter-packet time gaps (Fig. 1). A chirp is thus a sequence of packets with decreasing inter-packet time gaps.

The average rate r_{avg} of a whole chirp is the sum of all the packet sizes within it, divided by the sum of all the gaps between packet departure times, i.e. divided by the duration of the whole chirp. When used for congestion control, all data packets are within chirps. But the average rate of each chirp is arranged to track the intended sending rate of the congestion control algorithm.

Chirping should guarantee that the impact of probing for higher rates on the network is limited. Packets with lower and higher rates than the chosen average rate can be sent over a brief duration. Thus one single chirp can probe for a wide range of possible sending rates. The range of rates probed for in a certain time can be controlled by the spread of the inter-packet gaps, and the number N of packet in a chirp.

By monitoring the relative queuing delays of every packet in a chirp the available bandwidth can be estimated. The nominal queuing delay of packet n is calculated using its sending and receiving timestamps, respectively ts_{snd} and ts_{rcv}, as follows:

$$q_n = ts_{rcv} - ts_{snd}. \quad (1)$$

The nominal queuing delay includes propagation delay that stays constant for long periods or varies only slowly. We are only interested in the growth in queuing delay between packets $\Delta q_n = q_n - q_{n-1}$, which cancels out any constant offset in each q_n. If the sending rate of a sequence of packets exceeds the available capacity of the bottleneck link, the packets will experience increasing delay as the queue in the network device grows. All subsequent packets of a chirp will queue up behind each other (self-congestion) and one-way delay will continue to increase.

Fig. 2 shows typical observations of queuing delay taken from one of our simulation runs. It shows persistently increasing values at the end of a chirp, starting from a certain packet. To a first approximation, the sending rate of this packet reveals the available capacity. The rest of the diagram and a better approximation will be explained later.

Thus the available capacity is not only estimated based on the queuing delay measurement of a pair of packets but on several in a row. Moreover, those delays are self-induced which means that several packets are sent at a too high rate, to deliberately build up a queue for a short period of time. Subsequently this characteristic delay signature can be correlated with the pre-set inter-packet gaps sent-out. In contrast to other delay-based estimation mechanisms the use of self-induced congestion should be more robust to noise.

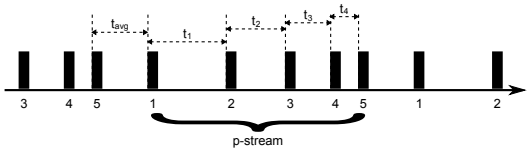

Figure 1: Multi-rate probe streams with N=5.

The instantaneous available bandwidth can change over the duration of a chirp, as competing flows vary their rate. The pathChirp paper proposes a way to estimate a weighted average of the available bandwidth over the duration of each chirp. Instantaneously lower available capacity is detected as an increase in queuing delay in the middle of a chirp, which then decreases before the end (labelled *excursion* in Fig. 2). To filter out noise, a such an excursion is included in the available bandwidth estimation if it follows an increasing trend for at least L packets. In addition, a change in queuing delay is only counted as part of an increasing trend if it is larger than $1/F$ of the maximum rise in queuing delay in the current excursion. Following tests, the defaults proposed by pathChirp for these variables are $L = 5$ and $F = 1.5$. For the final bandwidth estimation the average of the bandwidth estimation of each excursion weighted by its duration is used. The pathChirp paper admits that more sophisticated approaches may be possible to improve accuracy but initially we re-use their simple algorithm.

3. A CHIRPING IMPLEMENTATION IN THE LINUX KERNEL

In order to realise chirping within TCP congestion control, we implemented a RAPID-like congestion control in

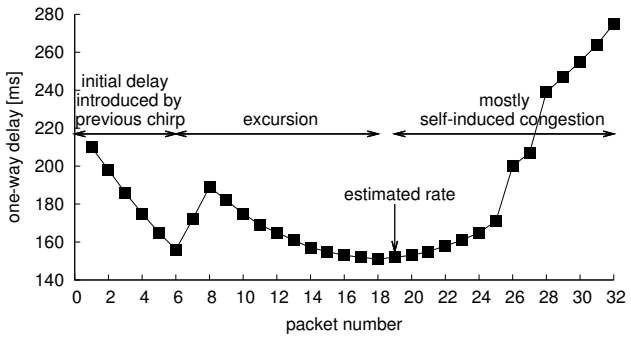

Figure 2: Queuing delay of a 32 packet chirp with TCP cross traffic.

the Linux kernel version 2.6.26. This paper focuses on the implementation of chirping as a building block for congestion control based on per-packet send-out timing. We focus less on the RAPID rate control algorithm itself, which researchers might choose to swap out for their own algorithm. The following four function blocks are needed to realise chirping within a congestion control protocol:

1. **Feedback for one-way delay measurement:** The protocol extensions needed and different solution for deployment are discussed in the following Subsection.

2. **Rate estimation:** The algorithm to evaluate the feedback information of one chirp based on pathChirp. Pseudo code of the implemented algorithm is attached in Appendix B.

3. **Rate adaption:** The chosen congestion control algorithm evolves the average sending rate of the next chirp based on its own algorithm using the available capacity estimated by the previous chirp. Furthermore CWND should also be updated to a value that allows the intended number of packets to be sent in one RTT.

4. **Inter-packet gap calculation:** To arrive at the chosen average sending rate the inter-packet gaps for the next chirp need to be recalculated. We developed our own algorithm in order to simplify kernel implementation (see §3.3).

§3.2 gives an overview of the timing framework for sending out packets and the structure of the implementation as congestion control kernel module giving the interdependencies between the components.

3.1 Sender-side Delay Measurement based on TCP Timestamp Option

The precision of our measurements does not rely on the time-gaps between packets conforming exactly to the precalculated chirp time-gaps. Instead, we timestamp a packet when it is actually sent and use this information for any further calculations. So even if a packet experiences delay elsewhere in the lower layers of the sender, the algorithm is designed to be robust, as long as all the actual sending gaps decrease within a chirp.

We chose to locate all timing analysis with the timing control that must naturally sit at the sender-side. We only gave the receiver the task of feeding back time stamp information for the delay measurements. We adopted TCP timestamps (TS) for this purpose, which are activated by default in the Linux kernel and in most modern operating systems. Table 1 shows the structure of the Option header which is added to the TCP headers in either direction. This TCP option provides a 32 bit TS Value field, which is supposed to carry a packet send-out timestamp. This gets echoed by the receiver using the TS Echo Reply field in the other direction along with the sending TS Value of the ACK itself. By comparing an echoed timestamp with the current system time, the sender can estimate the RTT.

Table 1: TCP Timestamp Option header.

Kind=8	10	TS Value (TSval)	TS Echo Reply (TSecr)
1	1	4	4

When (mis)using the TCP Timestamp Option to estimate the one-way delay, the send-out timestamp in the TS Value field of the ACK is compared to the echoed timestamp of the sender, as also used by TCP-LP [11]. The time gaps between sent packets are reconstructed from the echo timestamps of the current ACK and the previous ACK.

As chirping is only interested in changes in delay, clock synchronisation is irrelevant. But the timestamps of sender and receiver do need to use the same time resolution. The TS Echo Reply is actually specified to just echo the 32 bits in the TS Value no matter what those bits mean. Thus the sender needs to know the receiver's time resolution. In the Linux kernel TCP timestamps normally have millisecond resolution. To recognize increases in delay in a high-speed scenario this resolution is not sufficient. Thus to use chirping with TCP Timestamps in the Linux kernel a higher resolution and some kind of resolution negotiation are needed. The Linux kernel provides so-called high resolution timers (hrtimers) with nanosecond resolution. This gives sufficiently small inter-packet gaps for today's network speeds and negotiation would allow scaling to higher speeds in future.

To estimate one-way delay we need to associate the sent and received timestamps of one packet together. But the delayed ACK mechanism [3] confuses matters. Missing ACKs are not a problem—rate estimation could be done with only the ACKs in a chirp that are available. The problem with delayed ACK's is that the TS Echo Reply is supposed to reflect the TS value of the oldest packet it acknowledges, not the most recent (in order to give a conservative estimate for the Retransmit Time Out). Then the Echo Reply timestamp in an ACK is typically not the sending time of the packet that triggered the ACK, but that of an earlier packet instead.

Delayed ACKs are on by default in the Linux kernel and no protocol is available for a TCP sender to get the receiver to deactivate them. In our implementation we have manually deactivated delayed ACKs receiver-side. But if chirping proves useful, a negotiation protocol at flow start could turn off delayed ACKs.

However, given delayed ACKs serve a purpose, rather than deactivating them it would be better to negotiate alternative timestamp semantics. Ideally for more precise one-way delay measurement, as well as a sender timestamp, the receiver would timestamp the packet on reception and echo both sent and received timestamps in an ACK. This would

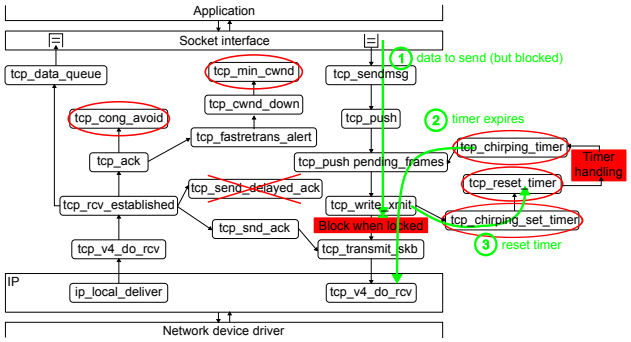

Figure 3: Changes in the TCP network stack.

intrinsically associate send and receive timestamps for the same datagram in one ACK. Experiments using our testbed to investigate how much this would improve bandwidth estimation are work-in-progress.

In our current implementation we remember the start of every chirp to keep chirp analysis synchronised even when e. g. ACKs are lost. To store less state at the sender in future we would like to attach this information—a chirp ID—to the packet header itself. One possibility is to exploit the opaque semantics of the 32 bit TCP timestamp option TS Value field. If we placed a chirp ID in this field, it would be blindly echoed at receiver-side. The alternative of a new TCP option would be cleaner, but would raise deployment issues.

Two of these new protocol designs—timestamp resolution and disabling delayed ACK's—could be implemented with negotiation during the connection handshake. Adding a chirp ID and adding a receive timestamp require further more considered protocol design.

To enhance accuracy, hardware time-stamping can be used. With Hardware timestamping the TSval field of the TCP Timestamp Option header is rewritten by the network interface device just before sending the packet. The same mechanism is often used with IEEE 1588-2008/IEEE 802.1AS Time Synchronization Messages which are already supported by a large number of network cards. The Linux kernel provides an interface to activate hardware timestamping.

3.2 Implementation Structure

As intended by the Linux kernel design we implemented the chirping algorithms based on the kernel congestion control module interface. Chirping requires controlled timing of each packet send-out time, so we introduced an additional TCP timer by using the kernel TCP timer framework. All changes are displayed in Fig. 3. When new data are available the send-out is blocked (1) until the chirping timer expires (2). Whenever a packet is sent, the next inter-packet gap is obtained to reset the timer (3). As every single packet sent-out is triggered by a timer interupt this design will lead to an increased number of context switches between user and kernel space that can slow down the system. We plan to further investigate this and the impact of other processing delays in a high-speed testbed. But we assume that in future the send-out timing can be realized completely in hardware if the use of chirping for congestion control can be shown as a promising approach.

The simplified pseudo code for the three methods implemented in the congestion control module realising the chirping algorithm is displayed below. `cong_avoid` and `min_cwnd` are methods already intended for congestion control by the Linux kernel module interface. With `reset_timer` we extended the congestion control module interface with a new method. `min_cwnd` is called when a loss event occurs. In this case we halve the rate, recalculate the inter-packet gaps and start a new chirp. The mechanism to ensure that this is only done once for every recovery event is not shown in the pseudo code.

`cong_avoid` is called for every ACK packet received as it was originally intended to open the CWND. We use it to store the chirping feedback information and adapt the average sending rate for the next chirp when feedback information of a whole chirp is available. First, the current available rate is estimated by *estimateRate()*. Then the congestion control algorithm to determine the new average sending rate is called. In a third step the new inter-packet gaps are calculated.

Finally, this new send-out pattern will be adopted in `reset_timer()` in the next chirp. `reset_timer()` is called whenever a packet is sent out. It returns the next value that the chirping timer should be set to.

Algorithm 1 Chirping congestion control module.

cong_avoid(): {called for every ACK}
remember one-way delay
remember reconstructed send-out inter-packet gap
update CWND
if last packet of chirp **then**
  $gap_{est} = estimateRate()$ {est. available bandwidth}
  $gap_{avg} = adaptRate(gap_{est})$ {new average rate}
  calculate new inter-packet gaps with Eqn. 3
end if
reset_timer(): {called for every data packet send}
if last packet of chirp **then**
  start new chirp
  remember/increase chirp identifier
end if
return next inter-packet gap
min_cwnd(): {called for every recovery event after loss}
$gap_{avg} = 2 * gap_{avg}$ {= halve r_{avg}}
recalulate inter-packet gaps with Eqn. 3
start new chirp

3.3 Algorithm for Inter-packet Gap Calculation

Our implementation is fully based on inter-packet time gaps instead of rate. Initially we assume equal sized packet which is usually the case when sending a stream of data without blocking. If packets cannot be sent continuously, because the application is blocking for data, usually the available bandwidth cannot be fully used anyway and bandwidth estimation based on chirping might not be appropriate.

To keep the Linux implementation as simple as possible, the chirp size N must be set to a value that is an integer power of two. Initially in our implementation it is hard-coded at 32 ($= 2^5$), given experiments in [9] recommended $N = 30$. To simplify kernel calculation effort, we calculate the time gap sizes using a harmonic progression of rates of slightly wider range than the geometric progression of [9].

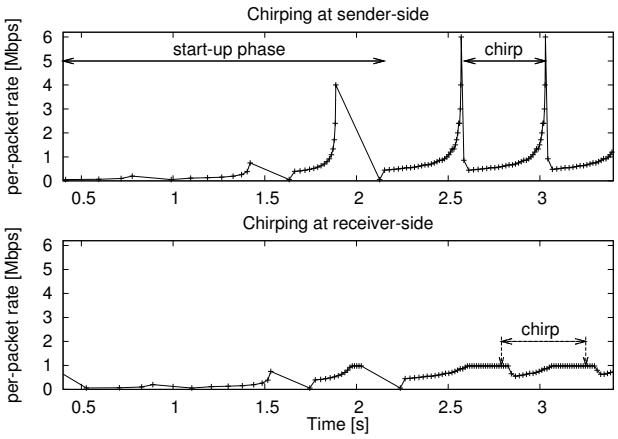

Figure 4: Per-packet rate of one continuous RAPID transfer with 32 pkt/chirp over an 1 Mbit/s link.

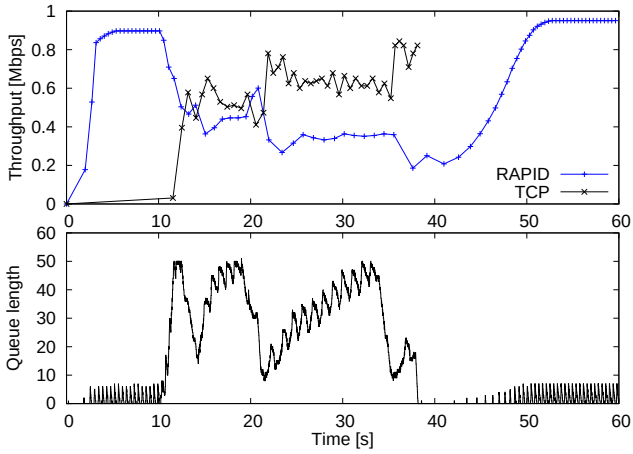

Figure 5: Throughput of RAPID and TCP cross traffic starting at 10 seconds.

Then the gap size before the i-th packet can be calculated by the simple linear decrease of

$$gap_i = gap_{i-1} - gap_{step} = gap_{i-1} - \frac{2gap_{avg}}{N} \qquad (2)$$

Using this would cause the ith gap to be

$$gap_i = \frac{2gap_{avg}}{N}(N - i + 1) \qquad (3)$$

with $i = 1...N-1$ and $gap_0 = gap_{avg}$. With this formula pairs of gaps from each end of the chirp can be regarded as grouped to $2*gap_{avg}$ besides gap_1 and gap_2 as we do not use the maximum range for the high rates. Thus this calculation leads to a slightly lower average rate than the estimated one and to a probing range from $\frac{1}{2}r_{avg}$ to $\frac{N}{4}r_{avg}$. An example for the resulting per-packet rate is shown in Fig. 4. The upper diagram shows the per-packet send-out rates of chirps of a RAPID connection probing around the bottleneck capacity of 1 Mbit/s on an empty link with a probing range of 0.5–6 Mbit/s. Fig. 4 also shows the Slow-Start-like phase proposed for RAPID congestion control. Details of how RAPID uses chirping during SlowSstart are relegated to Appendix A.

4. PRELIMINARY RESULTS

We have been running simulation with the IKR Simulation Library [7] and the Network simulation Cradle (NSC) [8]. This enabled us to use the kernel code within simulation components. Today, the NSC can handle Linux kernel code only up to version 2.6.26. We are planning to port our implementation to a newer version and run real-life tests in an environment up to 10 Gigabit/s. The current simulation results are limited in speed due to the restrictions of the timing in the NSC to a milliseconds resolution as hrtimers are not yet available. Due to this limited time resolution our scenarios are based on a 1 Mbit/s bottleneck link with larger access bandwidth of the sender to send packets with a probing rate up to 12 Mbit/s.
Fig. 4 also displays the receiver-side per-packet rate which is limited by the bottleneck bandwidth of 1 Mbit/s. Note with this simulations there is 100ms one-way transmission delay on the link. All packets sent with a higher rate than the maximum bottleneck speed build up a queue in the router. Even when the next chirp starts with lower per-packets rates it takes a while until the queue empties again. Those initial delays can also be seen in Fig. 2.

4.1 RAPID Congestion Control

Our investigation of the RAPID-like implementation discovered several problems in the design of RAPID congestion control. This knowledge can be used to get a better understanding of how to design a robust congestion control based on chirping.

RAPID congestion control directly uses the available bandwidth estimated by a previous chirp as r_{avg}. In common with many purely delay-based approaches, RAPID cannot compete for an equal share of capacity with loss-based approaches, such as standard TCP. By design RAPID only aims to scavenge any bandwidth they are not already using. Nonetheless, RAPID does build up some additional short-time delay in the router queues, which influences the TCP flow throughput.

For our implementation, we propose that all inter-packet gap sizes, which have been calculated with nanosecond resolution as a preparation for a high-speed testbed usage, should be rounded to the next higher value when setting a millisecond resolution timer for NSC usage. This will slightly underestimate the available rate but avoids queue build-up.

In the general case when a queue builds up because of permanent overestimation this will not be recognised by the delay measurements within a chirp, as the queue stays constant or decreases while sending packets with lower rates. Thus RAPID congestion control will not recognise its own impact on long-term queuing until a loss occurs. We expect a more sophisticated solution can be found here to get close to the right value without risking overestimation.

Regarding the case in RAPID design where multiple RAPID flows are competing, there needs to be some additional mechanism to converge to the intended bottleneck shares. Without such a mechanism the RAPID flow which has already achieved a larger sending rate will get earlier notification if spare capacity becomes available. In [9] it is proposed to equalise the rates to which RAPID senders converge by

introducing a parameter τ, as given in Equation 8 in Appendix A.

This is supposed to lead to an equal rate in the same amount of time (τ ms) for competing RAPID flows with different starting rates. Of course RAPID can only converge to an equal rate distribution if τ is set to the same value for all competing RAPID connections. This is very sensitive to variations in different implementations. Furthermore, if τ is larger than the time needed to send one chirp, the proposed formula will allow an increase larger than the estimated available rate. This will disturb competing flows strongly. Consequently in our simulation in a low speed scenario this approach led to very large inter-packet gaps and respectively a very low average rate of the competing flow. In fact one RAPID flow ended up sending only 32 packets over several seconds as it could not adapt within one chirp.

Fig 5 shows the throughput of a RAPID connection and a TCP connection competing for 1 Mbit/s bottleneck capacity with a drop-tail queue of 50 packet queue size and a base delay of 200 ms per round trip. The TCP connection started 10 seconds after the RAPID connection and goes on until 20 Mbit of data has been transmitted. Due to the introduction of the intra-protocol convergence factor τ (here set to 268ms for implementation reasons) the RAPID flow does not back off completely but whenever a loss event occurs the bandwidth share between the TCP and the RAPID flow changes. After termination of the TCP connection, RAPID converges slowly to the maximum rate. Without smoothing by using the τ value, the results are quite unpredictable.

We recommend that all implementations should not be require identical parameter values if dependencies on network characteristics exist, but instead the parametrisation should be made adaptive. We also argue that RAPID congestion control should be characterised as a scavenging approach emulating less than best-effort service, rather than an algorithm intended to compete on a par with TCP congestion control. A more scalable congestion control should not only rely on chirping information but use this fast feedback information to adapt the increase of the sending rate more appropriately to the actual network state. The rate adaption itself still needs to follow a scheme that can lead to convergence in capacity sharing with transfers using different congestion control mechanisms or different parametrisations.

4.2 Next Steps

Future work is still required to investigate the impact of short term probing delays on the queue burstiness and the influence of a large aggregation of probing chirps on the base queue length. We expect that every single chirping transfer should still be able to correlate its own send-out pattern with the received delay measurements. In return the chirping information can be used to reduce the congestion control overshoot and respectively the maximum queue length while still providing a fast acceleration in high speed environments.

Using chirping with retransmission or if too few packets are available needs further investigation, as well as issues on reordering, idle periods or incomplete feedback information. Furthermore, adaptation of the chirping parameters themselves, e.g. a selection of the chirp size N based on the sending rate or observed changes in rate, could lead to higher accuracy of the chirping information.

5. RELATED WORK

Next to RAPID congestion control, NF-TCP [2] is another proposal to use chirping for congestion control in combination with ECN-based congestion avoidance techniques. Instead of sending the data packets grouped in probing streams, NF-TCP uses separate probing packets to define a new sending rate. In return NF-TCP backs off early based on low-priority ECN-markings to remain TCP-friendly. The implementation of chirping was realized in user-space.
Regarding the implementation of rate-based congestion control approaches [4] proposes the reintroduction of ACK-clocking and window-based rate adaption to overcome implementation complexity. We cannot use this approach as an exact per-packet timing is needed. A similar approach to our implementation on timer-based packet spacing can be found for TCP Pacing [12] [10] [1]. PSPacer [15], as another reference to realize inter-packet spacing, follows a different approach, where so-called PAUSE packets are interleaved to increase the sending gaps between two data packets. [13] uses TCP Pacing and the packet-pair technique to improve the TCP start-up. PacedStart [6] introduced a sent-out packet spacing in Slow-Start for bandwidth estimation. Hybrid Slow Start [5] uses the fact that packet bursts in Slow-Start get paced out in the network by monitoring the ACK train duration.

6. CONCLUSION AND OUTLOOK

In this paper we presented what is needed to implement chirping as a building block for congestion control in the Linux kernel. The nanosecond resolution provided by kernel hrtimers is sufficient for today's speed, but initial negotiation about timer resolution and either receive timestamping or non-delayed ACKing needs to be added to the protocol design.

We ran initial experiments using our chirping implementation driven by the RAPID congestion control algorithm. We can already conclude that chirp parameters ought to adapt to prevailing conditions and that the available bandwidth estimates from chirping should be used in addition to other network state information. We have also noted that RAPID congestion control would be better characterised as a scavenger protocol, as it is not designed to take capacity share from protocols like NewReno that are loss-based.

We are planning to further study the impact of chirping on burstiness and queue length in a high-speed testbed. This will also provide insight into the dependency of the accuracy of the chirping information on timestamp resolution. As chirping provides faster feedback than today's solely loss-based mechanisms, our goal is to use the available bandwidth estimates to enable more scalable rate adaption with minimal overshoot.

7. REFERENCES

[1] A. Aggarwal. Understanding the performance of tcp pacing. In *Proc. 2000 IEEE INFOCOM Conference*, pages 1157–1165, 2000.

[2] M. Arumaithurai, X. Fu, and K. K. Ramakrishnan. NF-TCP: Network Friendly TCP. In *17th IEEE Workshop on Local and Metropolitan Area Networks (LANMAN 2010)*, May 2010.

[3] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, IETF, October 1989.

[4] S.-H. Choi and M. Handley. Designing TCP-Friendly Window-based Congestion Control for Real-time

Multimedia Applications. In *Inproceedings of PFLDNeT*, 2009.
[5] S. Ha and I. Rhee. Hybrid Slow Start for High-Bandwidth and Long-Distance Networks. In *PFLDNeT'08*, 2008.
[6] N. Hu and P. Steenkiste. Improving TCP Startup Performance using Active Measurements: Algorithm and Evaluation. In *Proc Int'l Conf on Network Protocols (ICNP'03)*. IEEE, November 2003.
[7] IKR, University of Stuttgart. IKR Simulation and Emulation Library, December 2009. http://www.ikr.uni-stuttgart.de/Content/IKRSimLib/.
[8] S. Jansen and A. McGregor. Simulation with Real World Network Stacks. In *Proc. Winter Simulation Conference*, pages 2454–2463, 2005.
[9] V. Konda and J. Kaur. Rapid: Shrinking the congestion-control timescale. In *INFOCOM 2009, IEEE*, pages 1 –9, apr. 2009.
[10] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge. Paced TCP for High Delay-Bandwidth Networks. In *IEEE Workshop on Satellite Based Information Systems*, 1999.
[11] A. Kuzmanovic and E. W. Knightly. Tcp-lp: low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 14:739–752, August 2006.
[12] D. Lacamera. TCPPacing. http://danielinux.net/index.php/TCP_Pacing, October 2010. Website.
[13] C. Partridge, D. Rockwell, M. Allman, and R. K. J. P. Sterbenz. A Swifter Start for TCP. Technical report, BBN Technologies, 2002.
[14] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell. pathchirp: Efficient available bandwidth estimation for network paths. In *In Passive and Active Measurement Workshop*, 2003.
[15] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. In *In Proceedings of PFLDNet 2005*, 2005.

APPENDIX

A. RAPID CONGESTION CONTROL

RAPID congestion control is designed to acquire "the available bandwidth within only a few RTTs". Furthermore, it aims to "achieving fairness among co-existing RAPID transfers" and "remaining TCP-friendly" [9]. The following enumeration will give an short overview about the proposed alogorithm:

1. **Rate-based packet transmission (at the sender):** Send multi-rate probe streams of N packets with decreasing inter-packet gaps of $t_i = P/r_{i-1}$ with $t_0 = t_{avg}$ and
$$r_{avg} = \frac{N-1}{\frac{1}{r_1} + \frac{1}{r_2} + ... + \frac{1}{r_{N-1}}} \quad (4)$$
for all $i > 1$, $r_i > r_{i-1}$.
2. **Available Bandwidth (AB) estimation analysis (at the receiver):** Implement pathChirp bandwidth estimation based on self-induced congestion. If q_i is the queuing delay experienced by the i-th packet and
$$q_k = 0, \quad \text{if } r_k \leq AB \quad q_k > q_k - 1, \text{otherwise,} \quad (5)$$
the available bandwidth estimation *ABest* will be given by the small rate where self-incuded congestion can be observed. *ABest* of the most recent p-stream will be forwarded to the sender in every next ACK.
3. **Transmitting in a non-overloading responive manner:** Set $r_{avg} = ABest$ for next p-stream to not exceed bottleneck capacity but simultaneous probing for de-/increase.
4. **Setting $[r_1, ..., r_N - 1]$ (speeding up the search process):** Implement multiplicative relation
$$r_i = m^{i-1} * r_1 \quad (6)$$
with $1 < i < N$ and
$$r_1 = \frac{m^{N-1} - 1}{(N-1)(m-1)m^{N-2}} r_{avg} \quad (7)$$
with $N = 30$ and $m = 1.07\%$. These values yield to a range of probing rates from $r_1 \approx 0.45 * r_{avg}$ to $r_{N-1} \approx 3.22 * r_{avg}$.
5. **Achieving a Quick-yet-Slow-Start:** Send only a single p-stream over the first four RTT's with $N = 2, 4, 8, 16$ and $m = 2$ and initialize r_{avg} to 100 Kbps. This can probe for up to 3342 Gbps in 4 RTT's (and is not more aggressive than other Slow-Start implementations).
6. **Dealing with packet loss:** Reduce r_{avg} by multiple of 0.5 after loss recovery.
7. **Bias due to rate-proportional feedback frequency:** Equalize the rate at which RAPID senders converge to *ABest* by
$$r_{avg} = r_{avg} + \frac{l}{\tau}(ABest - r_{avg}) \quad (8)$$
where l is the duration of the most-recent p-stream, which is given by: $l = \frac{N*P}{r_{avg}}$. τ represents a common time interval over which any RAPID flow should converge to an increase. τ is proposed to be set by default to 200ms.

B. AVAILABLE BANDWIDTH ESTIMATION

The implementation of the bandwidth estimation (based on pathChirp [14]) can roughly be described by the following pseudo code where **q** a vector of queuing delay of a chirp and **gap** a vector of the recalculated inter-packets gaps:

```
estimateRate(q, gap) {
for each packet do
  if q_i not increasing or less than q_max/F then
    remember as 'not an excursion'
    set new estimation gap_est to gap_i
    if previous excurison was smaller than L then
      remember all packet as 'not an excursion'
    end if
  else
    update q_max
  end if
end for
set gap_avg to gap_est
for each packet do
  if part of excursion then
    gap_avg += gap_est
  else
    gap_avg += gap_i
  end if
end for
gap_avg = gap_avg/N
}
```

This goes inline with the mechanims described in [14] but leads to any easier implementation.