PSIRP
Traffic and Congestion Control Implementation

TR10-0001

Document Properties:

Title of Contract	Publish-Subscribe Internet Routing Paradigm
Acronym	PSIRP
Contract Number	FP7-INFSO-IST 216173
Start date of the project	1.1.2008
Duration	33 months, until 30.09.2010
Document Title:	PSIRP - Traffic and Congestion Control Implementation
Date of preparation	17.09.2010
Author(s)	Ventzislav Koptchev, Kaloyan Petrov, Vladimir Dimitrov (IPP-BAS)
Responsible of the deliverable	Ventzislav Koptchev Phone: +35929796615 Email: vkoptchev@acad.bg; vgd@acad.bg
Target Dissemination Level[1]:	PU
Status of the Document:	Final
Version	1.01
Document location	http://www.psirp.org
Project web site	http://www.psirp.org

[1] Dissemination level as defined in the EU Contract:
PU = Public
PP = Distribution limited to FP7 participants
RE = Distribution to a group specified by the consortium
CO = Confidential, only allowed for members of the consortium

Production Properties:

Reviewed by:	George Parisis, Dirk Trossen

Revision History:

Revision	Date	Issued by	Description
0.01	2010-08-23	Kaloyan P.	Skeleton
0.02	2010-08-30	Kaloyan P.	Introduction and Conclusion
0.03	2010-09-15	Ventzislav K.	Implementation description
0.04	2010-09-16	Vladimir D.	Formatting, editing
0.05	2010-09-16	Kaloyan P.	Finished Introduction and Conclusion
0.06	2010-09-17	Vladimir D.	Draft.
0.07	2010-09-21	Ventzislav K.	Revised after the review
1.0	2010-09-24	Dirk T.	Finalized
1.01	2010-11-03	Vladimir D.	Cosmetic. Convert to PDF. Final.

This document has been produced in the context of the PSIRP Project. The PSIRP Project is part of the European Community's Seventh Framework Program for research and is as such funded by the European Commission.

Table of Contents
Table of Contents

Abstract

PSIRP is a project for researching the future Internet based on the publish/subscribe paradigm. It changes the focus from communicating endpoints to information being exchanged (between whoever can provide this information).

As a new architecture with its own characteristics, some form of congestion control was needed. We present a first attempt for such traffic and congestion control (TCC), which has been implemented as a software module for PSIRP Blackhawk prototype, implementing two stateless rate control mechanisms – sender and receiver controlled. Lost packet recovery is also implemented. As an addition, node caching was developed to increase network performance. The proposed TCC scheme is adopting the notion of a flow from current mechanisms in an information-centric environment. It is by no means intended to be THE mechanism for TCC but a starting point to evaluate this important area further.

1 Introduction

Congestion Control is an important part of every network architecture with shared resources and it can't be realized as a pure end-to-end function only. Congestion is an inherent network phenomenon and can only be resolved efficiently by some cooperation of end-systems and the network [Pap2009]. The function of TCC module in PSIRP is to increase utilization of network interfaces, prevent node overload and preserve stability of the network. Traffic control, congestion control, lost packet recovery and caching are all implemented in the TCC module and work together to achieve higher network utilization.

Lost packets are detected either by a skipped sequence number or exceeded packet expectancy timeout.

PSIRP provides the possibility to achieve packet level caching in the network by using packet Ids. The cache is node local and interface independent.

TCC was developed as a standalone software module based on the beta Blackhawk prototype. Notice that the current prototype does not support any transport mechanism for achieving the aforementioned goals.

2 Implementation

In addition to the existing *SUBDATA* packet type used in the official Blackhawk release by subscribers to request all publication packets, a *SUBDATACHUNK* request is implemented allowing subscribers to request specific publication chunks. The mechanism is used by the 'Lost chunk recovery', 'Data caching' and 'Subscriber controlled Traffic and Congestion Control (TCC)' mechanisms, as described below. The requested publication chunks are specified by their sequence numbers and are included in the *SUBDATACHUNK* packet payload as an array with the number of requested packets as the first element, followed by the sequence numbers of the packets.

2.1 Recovering from packet losses

Unlike in TCP, in the PSIRP paradigm there are no acknowledgement packets to aid with packet loss detection. However a lost packet can still be detected by its unique sequence number. For that purpose the publisher must send the packets in a natural sequence. In addition a packet expectancy timeout is set based on the round-trip-time (RTT) to a particular source, such that after the timeout is reached any missing packets are considered lost. In order to aid packet loss detection in *SUBDATACHUNK* requests, a 'flow' sequence number is added to the packet header where 'flow' stands for a given *SUBDATACHUNK* request. In general a *SUBDATACHUNK* packet is used to request a non-sequential subset of all publication chunks, which makes the 'flow' sequence numbers different from the publication sequence numbers. Unlike the publication sequence numbers, the 'flow' sequence numbers are sequential for the given *SUBDATACHUNK* request, with the last sent packet of the request having a number zero, allowing for lost packet detection within such requests.

The flow sequence numbers raise the following multicast issue. As already stated, these numbers need to be sequential for every request in order for lost packet detection to work. A common chunk requested by more than one subscriber will in general be part of a different chunk subset, thus having a different flow sequence number in the different replies. For example, let us suppose that a *subscriber1* requests chunks *2, 3, 6, 9* and a *subscriber2* requests chunks *4, 6, 10*. Assuming the publisher sends the chunks starting with the last, the flow sequence number of the common chunk 6 would be 2 in the first case and 1 in the second case.

Lost chunk recovery is implemented and relies on the following two functionalities:

- When the last packet of a given request (recognized by a zero 'flow' sequence number) is received, any missing packets from that request are considered lost and the subscriber sends a *SUBDATACHUNK* request containing only those.
- Cases when the last packet of a request is lost are handled by a program running in a separate thread. For each active publication an RTT is calculated when the first requested chunk is received, the program also keeps track of the system time the last packet was received. It checks all active publications at regular time intervals, the default value being one

second. If for an active publication it is determined that there are missing requested packets and the time elapsed since the last packet was received (or the active publication created) is greater than five times the RTT, all requested packets which are missing are considered lost and a *SUBDATACHUNK* request containing only those is sent. If necessary the process is repeated up to 10 times by default, after which if there are still missing packets of the given request, no further attempts are made and the active publication is freed.

There are three configurable parameters:

- the active publication checking interval. Default value 1 sec.
- packet timeout ratio relative to the round trip time. Default value 5.
- number of attempts to get the lost chunks. Default value 10.

2.2 Traffic and Congestion Control (TCC)

The goal of a Traffic and congestion control algorithm is to detect the congestion condition and reduce the rate at which packets enter the congested area. In the proposed TCC algorithm two congestion indicators are used – packet loss and node overload. For the rate control two independent mechanisms have been developed, which can be activated separately or in parallel.

- The first relies on stochastic fairness queueing (SFQ) and sending rate reduction of outgoing data using token bucket filters (TBF). The mechanism is applied at every node along the path. We refer to it as “sender controlled TCC”, the 'sender' being every node which forwards requested content. In this scheme the presence of congestion is communicated between adjacent nodes via special packets, *the choke packets*.
- The second mechanism - “subscriber controlled TCC” - relies on reduction of the rate at which the subscribers request data.

2.2.1 Sender controlled TCC

In this TCC version, the rate towards the congested area is controlled at the nodes, which send publication data. Unlike in TCP where the communication is of an end-to-end type and the client notifies the server to reduce its sending rate, here the rate can also be modified at any intermediate node, hence the name 'sender controlled' rather than 'publisher controlled'.

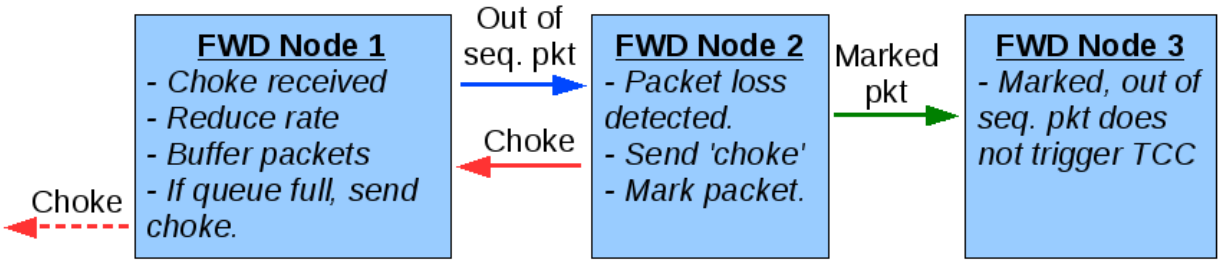

Figure 1: Sender controlled TCC

Every node divides traffic into separate **flows** based on packet FIds and publication meta-data and can trigger the TCC mechanism for a given flow if it receives an out of sequence data packet, indicating a packet loss in that flow. An out of sequence packet should only trigger the mechanism at the first node it is detected. For the purpose the first node, which detects the loss marks the out of sequence packet. Once marked, the packet will not trigger the TCC mechanism at subsequent nodes. A node that has reduced its sending rate due to congestion will receive data at a faster rate than its sending rate. In such conditions the data is buffered in the node's output queue. If the free buffer space gets low the node will signal the previous node, which in turn will reduce its sending rate and start buffering data. In this way the overload will be pushed back and spread over space, resulting in small peak overloads. This event, called *spatial spreading*, can provide the same amount of congestion buffer space, but in several nodes, reducing the probability of node overload. Thus if the bottleneck node is very close to the receiver, there are potential gains due to *spatial spreading* [Yun2004].

A node communicates a congestion condition to the previous node via a special *choke packet*. The flow sequence number field in the header of the choke packet is initialized to the flow sequence number of the packet, which triggered the TCC mechanism. The purpose of this is the following: since it takes time for a node to receive and react to a *choke packet*, a congestion detecting node will in general experience multiple packet losses and thus send multiple choke packets. The TCC mechanism should only react to the first received *choke packet* for a given congestion condition. It does so by keeping track of the flow sequence number of the last packet it has sent before reducing the sending rate. If an incoming *choke packet* has a flow sequence number larger than the number at which the TCC mechanism has reduced the sending rate, then the choke is a result of a congestion condition, which has already caused rate reduction and is neglected. *Choke packets* are only designed for adjacent nodes communication and are not forwarded. A *choke packet* is sent if either an unmarked out of sequence packet is received or the buffer space is getting low.

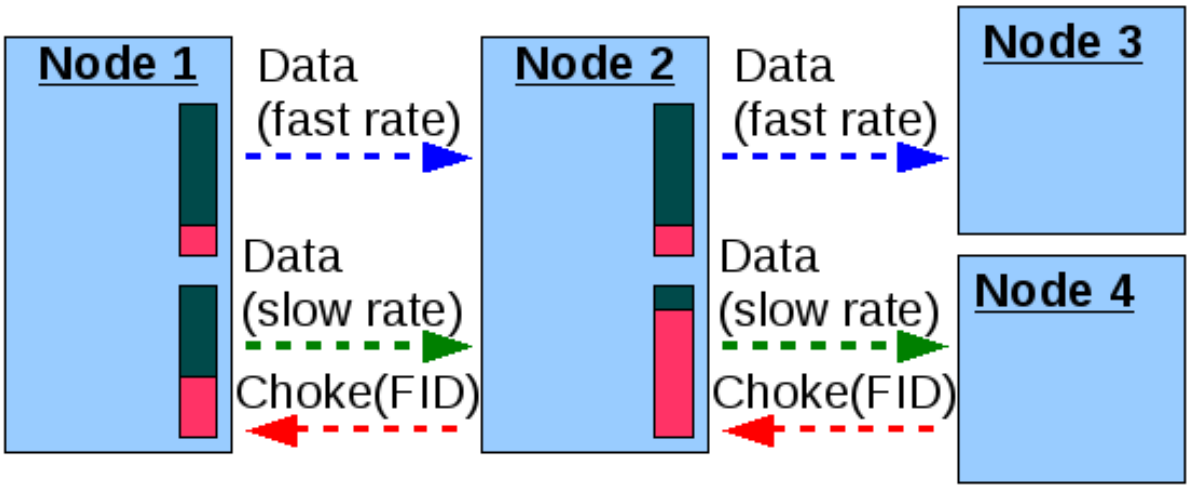

Figure 2: Flow separation in sender control TCC

All nodes create a separate outgoing queue for each flow. This allows for independent flow rate control. Moreover, the situation where congestion along one route would undesirably affect the rate towards other uncongested destinations (see

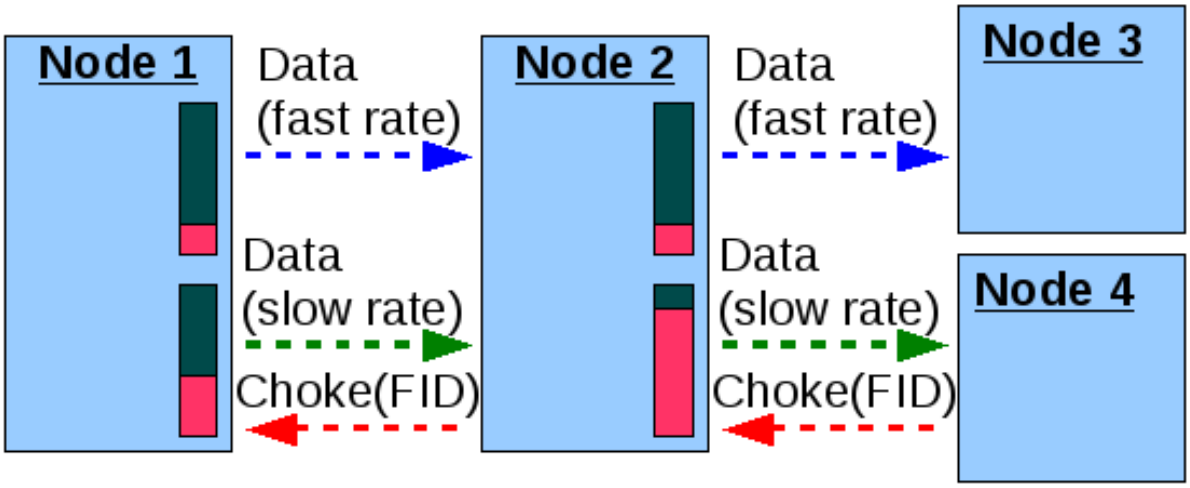

Figure 2) is also avoided. All queues are processed in a round-robin fashion and the queue rate control is implemented as a token bucket filter. In addition all nodes have a single input queue where all incoming packets wait to be processed with the goal of using this queue as an indicator of node overload.
When a node receives a *choke packet* for a particular flow, specified by the *FID* in the packet`s payload, it reduces the sending rate for that particular flow in a single step to a predetermined value. The rate then increases multiplicatively if no other *choke packets* concerning this particular flow are received.

The TCC module is implemented as a shared library written in C++ with a C interface:

- `void tbf_enqueueOut( const psirp_fid_t*  fid, if_list_item_t* iface_out, const void*  buf, size_t len, int flags)`

The first argument is used to classify traffic flows. It is followed by the output interface, the beginning of the packet, the packet length and the flags to pass to the *sendto()* system call when the packet is dequeued.

- `unsigned int tbf_congestionCheck(const psirp_fid_t* fid, char* classId)`

Checks if the output queue corresponding to the *FID,* provided as the first argument, is getting full. Returns a congestion status: 0 (no congestion), 2 (the queue is filled above 50%), 4 (the queue is full). The second argument is set by this method to the *ID* of the queue. The caller uses the *classId* to initialize the payload of the choke *packet*. A node that receives a choke packet uses the classId to modify the sending rate of the particular queue corresponding to the classId.

- `char* tbf_classify(const psirp_fid_t* fid)`

Returns the classId corresponding to the *FID*. Used for sending choke packets when the input queue is congested.

- `void tbf_reduceRate(char* classId)`

Reduces the sending rate of the corresponding queue.

The following parameters can be configured:

- output queue size – the size in number of packets of the output queues. There is one such queue for every flow at every node (default value 500).
- token bucket size – the size in number of tokens of the buckets used to control the rate at which packets are dequeued from the output queues (default value 10).
- initial congestion rate - upon receiving a *choke packet*, the receiver drops the corresponding sending rate in a single jump to a constant value determined by this parameter. The output rate of a queue is controlled with a sleep time interval between consecutive token additions to the bucket corresponding to the queue in question. This parameter specifies the sleep time in microseconds which is set immediately after receiving a choke packet.. Default value 1.000.000 microsececonds.

Note: In order to quicken the choke response at the receiver node, any existing tokens are removed from the corresponding bucket when a *choke packet* is received.

- sending rate increase factor - After receiving a *choke packet* followed by a sending rate drop, the rate increases multiplicatively by a factor determined by this parameter. The parameter is a division factor by which the sleep time (between consecutive token additions) is reduced with every consecutive token addition (default value 5).
- output queue threshold – the fraction in [%] of an output queue being full above which a *choke packet* is sent to the previous node (default value 50).
- flow inactivity check interval – the time interval in seconds at which all existing output queues in a node are checked for activity (default value 120).
- flow inactivity timeout – when performing an inactivity check, if the module finds an output queue which has been inactive (no sent packets) for a period longer than this value in seconds, the queue corresponding to this flow is deleted (default value 300).
- input queue size – the maximum number of packets that the input queue can buffer (default value 500).
- input queue threshold – the fraction in [%] of an input queue being full above which a *choke packet* is sent to the previous node (default value 50).

2.2.2 Subscriber controlled TCC

In this TCC version, which is of a TCP-like type in the sense that the TCC mechanism is always triggered by the "client" (subscriber), the rate towards the congested area is controlled indirectly through the number of chunks the subscriber requests at a time.

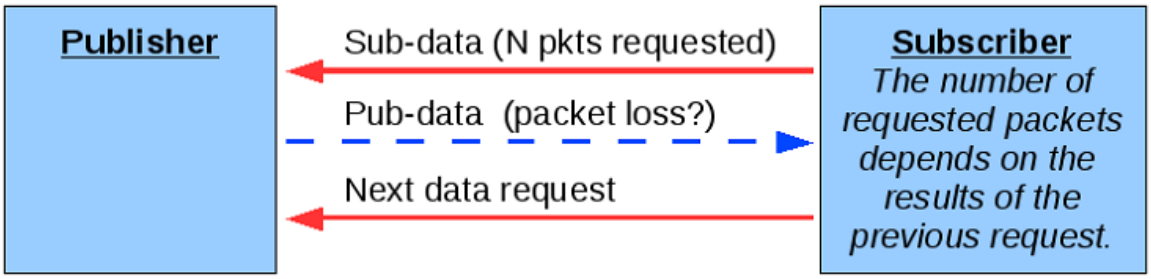

Figure 3: Subscriber-controlled TCC

The publication is obtained with a series of data requests containing a subset of all data chunks. A "slow start" algorithm is implemented by the subscriber starting to request three chunks. If no packet loss is detected the number of requested chunks increases by a factor of two with every consecutive request. In case of a packet loss, the number of requested chunks is reduced according to the percentage of lost packets in the previous request. Every consecutive request is triggered by the receipt of the last packet of the previous request, which incurs an RTT overhead in the overall publication reception time for every request, compared to an ideal case where all chunks are requested at once and there is no packet loss. Such overhead, however, could be reduced through interleaving traffic flows on the transport level. The following parameters are configurable:

- initially requested number of packets (default value 3).
- request rate increase factor. The number of requested packets increases by this factor with every consecutive request (default value 2).

2.2.3 Data caching

The caching mechanism consists of a single cache store per node servicing all network interfaces with the following configurable parameters:

- Check Interval - the time interval in seconds at which the cache is checked for old entries (default value60).
- Timeout - the time interval since a cache entry was entered or last used after which the entry is considered old and removed (default value5 minutes).
- Size - the size of the cache in number of packets (default value1000). If the cache size overflows, new entries are not stored until space is available.

The cache retrieval functionality relies on the *SUBDATACHUNK* type of request, through which specific publication chunks can be requested. Upon receipt of such a

request, a caching node checks its cache for the requested content. Any chunks found are sent back to the requesting node. To achieve this, the *FID* from the metadata header of the *SUBDATACHUNK* request which triggered the cache lookup is copied to the forward header FID field of the found cache entry. The returned cache entries for a given *SUBDATACHUNK* request are considered a separate flow and the flow sequence numbers of the entries are set accordingly. In case some of the requested chunks were not found in the cache, a modified *SUBDATACHUNK* request, containing only those chunks is forwarded towards the publisher. For that purpose, the initial *SUBDATACHUNK* packet is used with only the packet payload containing the requested chunk sequence numbers being modified. The process continues upstream until all requested chunks are delivered either by a caching node or by the publisher.

Currently, only caching nodes along the path from subscriber to publisher are potential data sources, i.e., implementing an opportunistic caching mechanism along the forwarding path from publisher to subscriber. This leads to the following limitation (see **Figure 4**).

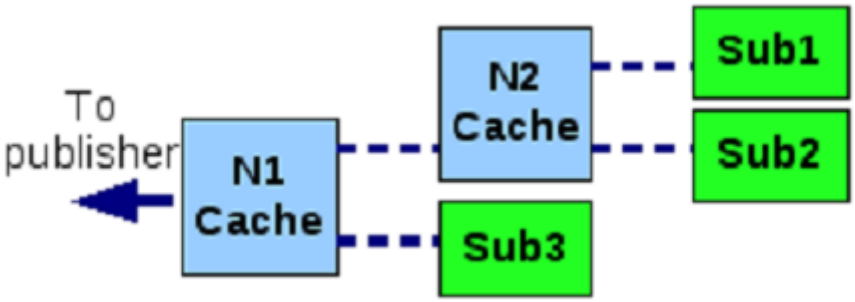

Figure 4: Caching Operation

If *Sub1* subscribes to a publication, the publication data will be cached in *N1* and *N2*. At some later time, before the cached data timeout is reached, *Sub2* subscribes to the same publication. The entire publication will be delivered from the cache of *N2* and the 'age' of the delivered cache items in *N2* will be reset back to zero. Later on, the timeout interval will be reached for *N1* and the publication data will be deleted. If at this point *Sub3* subscribes to the same publication, the data will be fetched from the publisher even though it is still available two nodes away at *N2*.

Data caching is implemented as a shared library written in C++ with a C interface:

- `void encache(pkt_ctx_t* pubdata)`

The argument is a pre-parsed clone of the original *PUBDATA* packet received by the node. The clone ownership is transferred to the cache. The clone is deleted by the cache cleanup thread when it is determined to be old.

- `pkt_ctx_t** getpubdata(pkt_ctx_t* subdatachunk, unsigned int* len)`

The first argument is the *SUBDATACHUNK* packet received by a node containing the sequence numbers of the requested chunks. The function returns an array containing all or a subset of the requested chunks and sets the second (output) argument to the length of the array.

- `cache_unlockItem(pkt_ctx_t* pubdata)`

Cache entries are returned in a locked state which prevents the cache clean up thread from removing them. This function is needed to release the locks after the entries have been processed.

The mechanisms covered in this report have been implemented on top of blackhawk-v0.3-beta2-20100507. Changes and additions to the original code have been commented with a name. Following is a short summary of the main modifications.

configure.ac
Added entries for module compilation and logging setup.

helpers/caching
Caching functionality implementation. Threads created:

- cache cleanup thread

helpers/randgen
Congestion simulation functionality implementation, allowing to randomly drop packets with rate-dependent probability. The probability function used is **`exp((x-A)/10-1)`**, where A is a rate threshold parameter and x is the current rate. See Figure 5

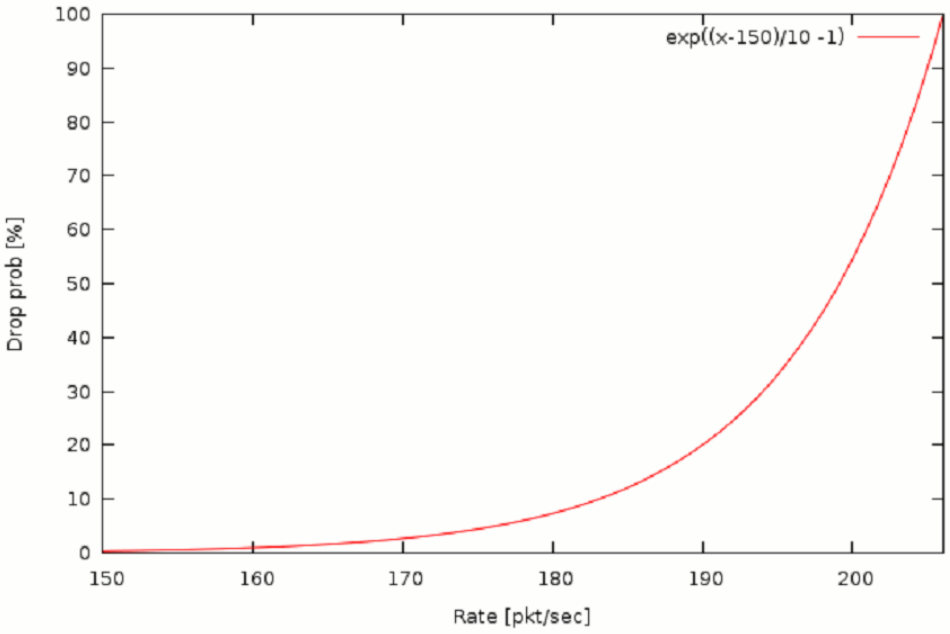

Figure 5 Packet drop probability distribution

helpers/tbf

Sender controlled TCC implementation. The TBF abbreviation came from 'token bucket filter' and was the original working title. At this point the naming needs revision since the module does a lot more than just rate control. Threads created:

- one token addition thread per flow[2]
- one dequeueing thread per flow
- thread for deleting inactive queues

helpers/laird/laird.py

SUBDATACHUNK request implementation allowing for a subscriber to requested specific chunks. Throughout the code the original rzv types related to the whole publication have been changed from PSIRP_HDR_RZV_SUBSCRIBE_DATACHUNK and PSIRP_HDR_RZV_PUBLISH_DATACHUNK to PSIRP_HDR_RZV_SUBSCRIBE_DATA and PSIRP_HDR_RZV_PUBLISH_DATA respectively. New rzv types PSIRP_HDR_RZV_SUBSCRIBE_DATACHUNKS and PSIRP_HDR_RZV_PUBLISH_DATACHUNKS have been added to denote SUBDATACHUNK requests.

netiod/Makefile.am

Additions to compile the tbf caching and randgen folder content.

netiod/netiod.c

1) options to switch on the following independent functionalities
   - -j – sender controlled TCC
   - -k – subscriber controlled TCC
   - -n – caching
   - -p – lost packet recovery
   - -d [rate threshold] – congestion simulation
2) the following threads are started in netiod.c
   - if sender controlled TCC is enabled
     - inbound packet queue thread. The entry function *pqueueInd()* is defined in psirpd_packet.c.
     - initializes the TBF module with a call to *initTBF()*. The module starts additional threads, described in the TBF section.
     - psirpd_out_q_send thread. In order to quicken the choke response at the choke receiving nodes, the call to *psirpd_out_q_send()* is moved to a separate thread when sender controlled TCC is enabled. The entry function *psirpd_out_q_send_thread()* is defined in psirpd_out_q.c.

   - if lost packet recovery or subscriber controlled TCC is enabled, the activepubs_timeout thread is started. As described in the lost packet recovery section, this is the thread which periodically checks all active publications at the subscribers.

[2] Since there are two dedicated threads per flow (one for de-queuing and one for token addition), this would make a lot of threads and will degrade the performance at some point. If the general idea of the described TCC turns out worthy of further development, this part of the code will have to be revised.

For each active publication the time period since the last received packet is calculated. If that period exceeds five times the round-trip time all missing chunks in the active publication which have been requested are considered lost and requested again. The entry function *psirpd_ipc_activepubs_timeout()* is defined in psirpd_ipc.c.

netiod/psirp_common_types.h
Inbound queue size and congestion threshold for the sender controlled TCC are defined here.

netiod/psirpd_fwd_bf.c
If sender controlled TCC is enabled the outbound queue congestion check is done here and if necessary a choke packet is sent.

netiod/psirpd_ipc.c
Implementation of the *SUBDATACHUNK* request and subscriber controlled TCC.

netiod/psirpd_net.c
Implementation of cache check, cache retrieval, forward cache request. Enqueueing of outgoing packets if sender TCC is enabled.

netiod/psirpd_packet.c
Added *SUBDATACHUNK* and choke packet handlers, data encaching, choke packet generation, inbound queue congestion check.

netiod/psirpd_rzv.c
Added *SUBDATACHUNK* and choke packet handlers.

3 Test Results

The modules have been tested using:

- virtual machine networks with various topologies and number of nodes, running on a 2 x CPU Intel Xeon E5430, 4 cores each, 2.66GHz, 16GB RAM.

- 4 real machines connected sequentially

- test publication size - 470 packets

As performance indicators we have monitored the overall publication reception time and the traffic overhead (repeated packets due to packet loss).

In the virtual machine setup the following is observed with sender-controlled TCC enabled. A time difference of over 10 seconds builds between the publisher and the next node after 200 sent packets, resulting in a situation where for a publication of maximum size (~470 packets), a packet loss at the 2nd node in the middle of the publication results in a choke being sent with a 10 seconds delay. When the publisher receives such a choke it has already sent all publication chunks. The sending rate is not altered since there is nothing more to send. Similar situation is observed when testing on real machines. The time difference that builds between the publisher and the next node after 200 sent packets is much smaller – around 0.2 seconds, but it is still large enough for the publisher to shoot all chunks before it receives any potential choke packets resulting from a packet loss at the second node in the middle of the publication or later.

With both sender-controlled and subscriber-controlled TCC enabled the publication is retrieved with series of data flows with RTT intervals in between. In this case the time difference that builds between the publisher and the second node is large enough for the publisher to send all requested chunks in the series before receiving a potential choke. Even though there is nothing left to send of the currently requested series, it is possible to reduce the sending rate for a following series after a choke is received. In our test however this has shown to be counter-productive as the RTT delay between the requested series has been enough to alleviate any congestion condition, rendering the rate reduction for the next series unnecessary.

The problem with the time difference described above is only observed between the publisher and the following node. According to our tests however, that is where most of the natural (non-artificially invoked) packet loss occurs. When inducing packet loss with the congestion simulation module at nodes further away from the publisher, the time delay has not been an issue and the sender-controlled TCC has worked effectively. The issue at the publisher hurts the overall performance of the sender-controlled TCC and in our tests the combination of subscriber-controlled TCC and packet loss recovery has outperformed it significantly when measuring overall reception time and traffic overhead.

4 Conclusion

This report laid out a first attempt to congestion control in a PSIRP environment. We adopted a basic AIMD type of algorithmic, supplemented by a simple caching mechanism. We implemented and evaluated the solution in an experimental setting.

We can preliminarily conclude that the AIMD type of algorithms for congestion control lead to oscillations during high traffic loads and should be further researched and optimized or replaced with better ones.

Further work should be done to evaluate if metadata could be used to optimize flow start-up and whether this can be achieve in the presence of caches in the network.

The possibility for middle node congestion reaction is a step forward to future CC alg., which will increase network stability by providing faster response time.

5 Terminology

AIMD	Additive increase multiplicative decrese
choke packet	Special packet used to communicate congestion condition between adjacent nodes.
classID	Based on packet FID and publication meta-data and used to divide traffic into separate flows.
FID	Forwarding Identifier
PSIRP	Publish Subscribe Internet Routing Paradigm
RTT	Round trip time
Sender TCC	A TCC mechanism in which the rate towards the congested area is controlled directly by the sending node.
SUBDATA	Type of request through which all publication data is requested at once.
SUBDATACHUNK	Type of request through which specific publication chunks are requested
Subscriber TCC	A TCC mechanism in which the rate towards the congested area is controlled indirectly by the subscriber.
TCC	Traffic and Congestion Control software module

6 References

[Kop2010] Koptchev V., V. Dimitrov, Traffic and Congestion Control in a Publish/Subscribe network. International conference CompSysTech'2010, Sofia, 17-18.06.2010. Published in "ACM International Conference Proceeding Series (ICPS)", Vol. 471,. Pages: 172-176. ISBN:978-1-4503-0243-2

[Pap2009] Open Research Issues in Internet Congestion Control, Papadimitriou, Welzl, Scharf, Briscoe, 2009

[Yun2004] Hop-by-hop Congestion Control over a Wireless Multi-hop Network, YungYi