CMPT 885: SPECIAL TOPICS: HIGH-PERFORMANCE NETWORKS

TCP-Friendly Rate Control: An Analysis
Fall 2003

FINAL PROJECT REPORT

Charu Jain
www.sfu.ca/~cjain
cjain@cs.sfu.ca

Abstract

New trends in communication, in particular the deployment of multicast and real-time
Audio/video streaming applications, Internet Telephony etc. is likely to increase the percentage of non-TCP traffic in the Internet. These applications use RTP over UDP and rarely perform congestion control in a TCP-friendly manner; they do not share the available bandwidth fairly with applications built on TCP, such as Web browsers, FTP, or e-mail clients. The Internet community strongly fears that the current evolution could lead to congestion collapse and starvation of TCP traffic. For this reason, TCP-friendly protocols are being developed that behave fairly with respect to coexistent TCP flows.
In this project, I present the implementation, simulation and evaluation of a simplified transport layer protocol that I have devised. The results obtained let me conclude that it is suitable for applications that use variable packet sizes but transmit at a constant rate (packets per second) like VoIP. Also, the protocol responds constructively to network congestion and is TCP-Friendly. I have used ns-2 tool support for this project.

Introduction

Most traffic in the internet uses TCP-based protocols such as Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), or File Transfer Protocol (FTP). However, the number of audio/video streaming applications such as Internet audio players, IP telephony, videoconferencing, and similar types of real-time applications is constantly growing, and it is feared that one consequence will be an increase in the percentage of non-TCP traffic. Since these applications commonly do not integrate TCP-compatible congestion control mechanisms, they treat the competing TCP flows in an unfair manner. Upon encountering congestion, all contending TCP flows reduce their data rates in an attempt to dissolve the congestion, while the non-TCP flows continue to send at their original rate. This highly unfair situation can lead to starvation of TCP traffic, or even to a congestion collapse, which describes the undesirable situation where the available bandwidth in a network is almost exclusively occupied by packets that are discarded because of congestion before they reach their destinations.

TCP Friendliness

In [1], non-TCP flows are defined as TCP-friendly when "their long-term throughput does not exceed the throughput of a conformant TCP connection under the same conditions."
TCP Friendliness for Unicast — A unicast flow is considered TCP-friendly when it does not reduce the long-term throughput of any coexistent TCP flow more than another TCP flow on the same path would under the same network conditions.

With the above definition, TCP friendliness ensures that coexisting TCP flows are not treated unfairly by non-TCP flows. Note, however, that this does not necessarily mean that all TCP and TCP-friendly flows on a bottleneck link receive the same throughput. Even competing flows that use only TCP for congestion control will often not receive the same amount of bandwidth. For example, TCP flows with different RTTs or different numbers of bottlenecks nodes will transmit at different rates.

Classification Schemes for TCP-friendly behavior

Window-Based vs. Rate-Based

One possible classification criterion for TCP-friendly schemes is whether they adapt their offered network load based on a congestion window or on their transmission rate. Algorithms that belong to the window-based category use a congestion window at the sender or at the receiver(s) to ensure TCP friendliness. Similar to TCP, each packet transmitted consumes one slot in the congestion window, while each packet received or the acknowledgment of a packet received frees one slot. The sender is allowed to transmit
packets only when a free slot is available. The size of the congestion window is increased in the absence of congestion indications and decreased when congestion occurs.
Rate-based congestion control achieves TCP friendliness by dynamically adapting the transmission rate according to some network feedback mechanism that indicates congestion. It can be subdivided into simple AIMD schemes and model-based congestion control. Simple AIMD schemes mimic the behavior of TCP congestion control. This results in a rate that displays the typical short-term saw tooth-like behavior of TCP. This makes simple AIMD schemes unsuitable for continuous media streams. Model-based congestion control uses a TCP model such as the one presented in [2] instead of a TCP-like AIMD mechanism. By adapting the sending rate to the average long-term throughput of TCP, model-based congestion control can produce much smoother rate changes that are better suited to the aforementioned type of traffic. Such schemes do not mimic TCP's short-term sending rate but are still TCP-friendly over longer timescales. However, the congestion control mechanism may not resemble TCP congestion control, and great attention has to be paid to the rate adjustment mechanism to ensure fair competition with TCP or other flows.

End-to-End vs. Router-Supported

Many of the TCP-friendly schemes proposed are designed for best effort IP networks that do not provide any additional router mechanisms to support the protocols. Thus, they can readily be deployed in today's Internet. These schemes are called *end-to-end* congestion control. They can be further separated into *sender-based* and *receiver-based* approaches.

In sender-based approaches the sender uses information about the network congestion and adjusts the rate or window size to achieve TCP friendliness. The receivers only provide feedback, while the responsibility of adjusting the rate lies solely with the sender.
Receiver-driven congestion control is usually used together with layered congestion control approaches. Here, the receivers decide whether to subscribe or unsubscribe from additional layers based on the congestion situation of the network. The design of congestion control protocols and particularly fair sharing of resources can be considerably facilitated by placing intelligence in the network (e.g., in routers or separate agents). Congestion control schemes that rely on additional functionality in the network are called *router-supported*. Particularly multicast protocols can benefit from additional network functionality such as feedback aggregation, hierarchical RTT

measurements, management of (sub) groups of receivers, or modification of the routers' queuing strategies. *Generic router assist* (GRA) [7], for instance, is a recent initiative that proposes general mechanisms located at routers to assist transport control protocols, which would greatly ease the design and implementation of effective congestion control protocols. Furthermore, end-to-end congestion control has the disadvantage of relying on the collaboration of the end systems. Experience in the current Internet has shown that this cannot always be assumed: greedy users or applications may use non-TCP-friendly mechanisms to gain more bandwidth. As discussed by Floyd and Fall in [1], some form of congestion control should be enforced by routers in order to prevent congestion collapse. The authors present router mechanisms to identify flows that should be regulated: for instance, when a router discovers a flow which does not exhibit TCP-friendly behavior, the router might drop the packets of that flow with a higher probability than the packets of TCP-friendly flows.

While ultimately fair sharing of resources in the presence of unresponsive or non-TCP-friendly flows can only be achieved with router support, this mechanism is difficult to deploy, since changes to the Internet infrastructure take time and are costly in terms of money and effort.

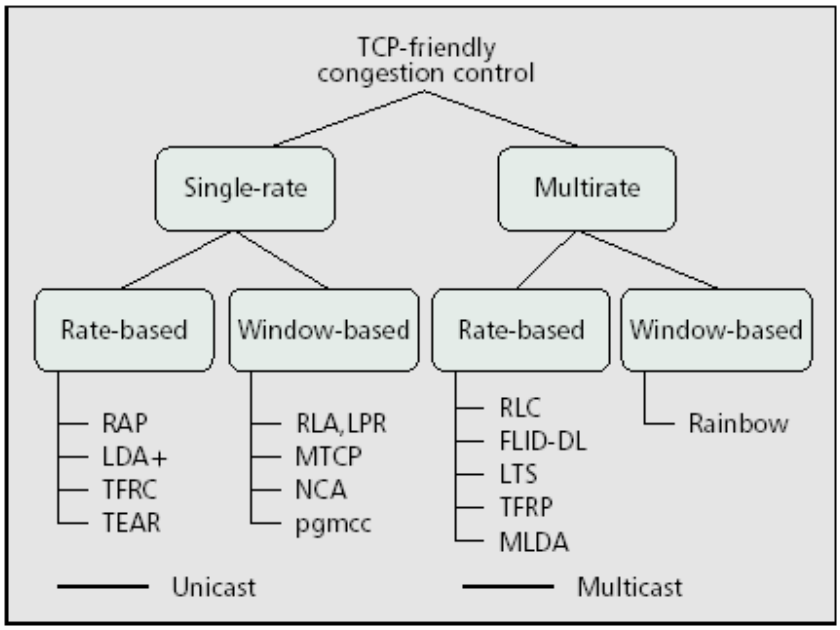


Figure 1. *A classification scheme for TCP-friendly protocols.*

An Overview of TFRC

TFRC is a congestion control mechanism for unicast flows operating in a best-effort Internet environment. It is reasonably fair when competing for bandwidth with TCP flows, but has a much lower variation of throughput over time compared with TCP, making it more suitable for applications such as telephony or streaming media where a relatively smooth sending rate is of importance.

The penalty of having smoother throughput that TCP while competing fairly for bandwidth is that TFRC responds slower than TCP to changes in available bandwidth. Thus TFRC should only be used when the application has a requirement for smooth throughput, in particular, avoiding TCP's halving of the sending rate in response to a single packet drop. For applications that simply need to transfer as much data as possible in as short a time as possible; TCP is recommended, or if reliability is not required, using an Additive-Increase, Multiplicative-Decrease (AIMD) congestion control scheme with similar parameters to those used by TCP.

TFRC is designed for applications that use a fixed packet size, and vary their sending rate in packets per second in response to congestion. Some audio applications require a fixed interval of time between packets and vary their packet size instead of their packet rate in response to congestion. The congestion control mechanism proposed by TFRC cannot be used by those applications; TFRC-PS (for TFRC-Packet Size) is a variant of TFRC for applications that have a fixed sending rate but vary their packet size in response to congestion.

TFRC is a receiver-based mechanism, with the calculation of the congestion control information (i.e., the loss event rate) in the data receiver rather in the data sender. This is well-suited to an application where the sender is a large server handling many concurrent connections, and the receiver has more memory and CPU cycles available for computation. In addition, a receiver-based mechanism is more suitable as a building block for multicast congestion control. A major advantage of TFRC is that is has a relatively stable sending rate while still providing sufficient responsiveness to competing traffic.

Problem Statement

Current TCP-friendly congestion control mechanisms such as those used in TFRC adjust the packet rate in order to adapt to network conditions and obtain a throughput not exceeding that of a TCP connection operating under the same conditions. In an environment where the bottleneck resource is packet processing, this is the correct behavior. However, if the bottleneck resource is bandwidth, and flows may use packets of different sizes, resource sharing then depends on packet size and is no longer fair. Now for some applications, such as Internet telephony, it is more natural to adjust the packet size, while keeping the packet rate as constant as possible.

TFRC-PS, that intends to provide a congestion control mechanism for VoIP-like applications, that vary their packet sizes and use a high constant rate of transmission, is still in it’s incipient stages of development and is an abstract concept.

Goal

To devise and implement a simplified protocol that is suitable for VOIP-like applications and at the same time responds constructively to Congestion.

Main Section

TFRC already implements a sophisticated mechanism for varying the rate in order to foster a constructive response to congestion. One possible approach would be to modify the mechanism

slightly and scale the packet sizes instead of the rate. Such an approach does not works and here is a brief explanation.

TFRC for VoIP-like applications

For its congestion control mechanism, TFRC directly uses a throughput equation for the allowed sending rate as a function of the loss event rate and round-trip time. In order to compete fairly with TCP, TFRC uses the TCP throughput equation, which roughly describes TCP's sending rate as a function of the loss event rate, round-trip time, and packet size. We define a loss event as one or more lost or marked packets from a window of data, where a marked packet refers to a congestion indication from Explicit Congestion Notification (ECN).

Generally speaking, TFRC's congestion control mechanism works as follows:

- ✓ The receiver measures the loss event rate and feeds this information back to the sender.
- ✓ The sender also uses these feedback messages to measure the round-trip time (RTT).
- ✓ The loss event rate and RTT are then fed into TFRC's throughput equation, giving the acceptable transmit rate.
- ✓ The sender then adjusts its transmit rate to match the calculated rate.

The throughput equation is:

$$X = \frac{s}{R\sqrt{2bp/3} + t_{RTO}\left(3\sqrt{3bp/8}\, p\,(1+32p^2)\right)}$$

Where:

X is the transmit rate in bytes/second.

s is the packet size in bytes.

R is the round trip time in seconds.

p is the loss event rate, between 0 and 1.0, of the number of loss events as a fraction of the number of packets transmitted.

t_RTO is the TCP retransmission timeout value in seconds.

b is the number of packets acknowledged by a single TCP acknowledgement.

Different TCP equations may be substituted for this equation. The requirement is that the throughput equation be a reasonable approximation of the sending rate of TCP for conformant TCP congestion control.

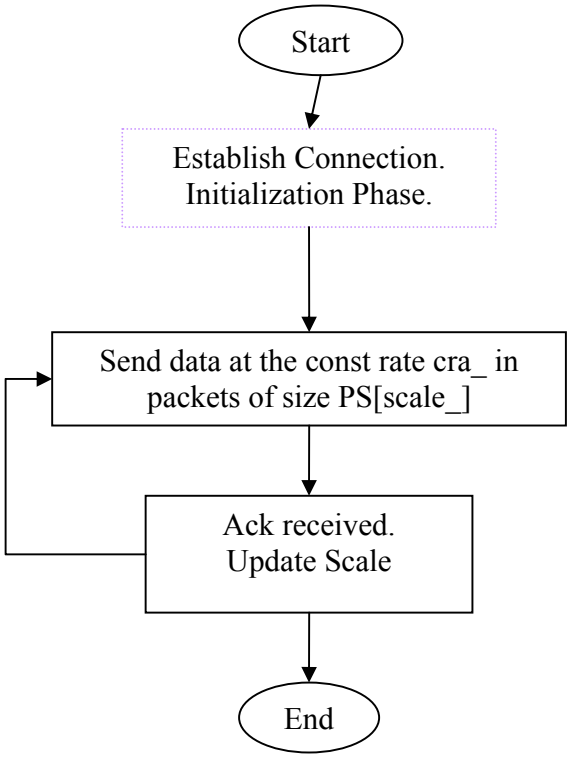
The loss event rate, p, is calculated at the receiver, by aggregating the packets lost in a one RTT into single loss event. For applications that transmit at a constant rate and vary packet sizes, the loss of a number of small packets in a single RTT would be aggregated into a single loss event, thus reducing the loss event rate, p. Clearly, this causes a bias towards sending small packets at a high rate. This is not at all fair since it results in lower bandwidth utilization. **Hence TFRC is not suitable for Internet Telephony applications.**

Packet-Size Scaling Protocol

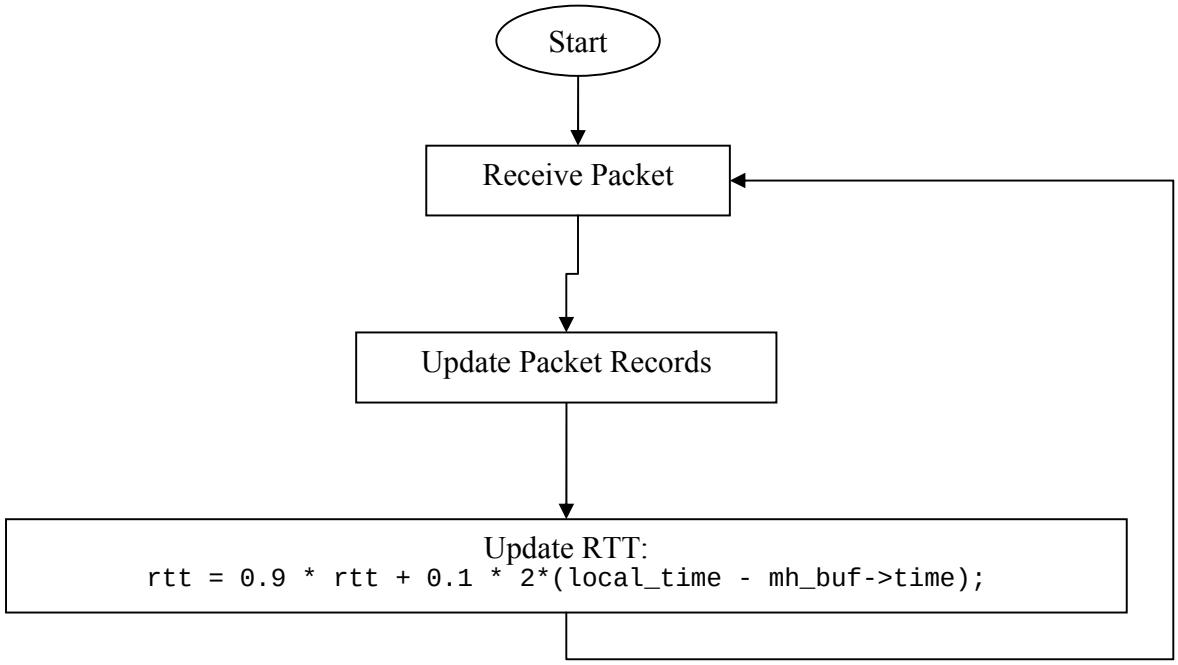
This is a simplified mechanism that employs a simple N-level packet scaling mechanism to respond to congestion. Basically this protocol sets a limit on the maximum packet size for an application, this limit is controlled by the present state of network congestion. The rate of transmission (Number of packets per second) remains constant and is decided before the transmission begins.

Detailed Flowchart (PSP Mechanism):

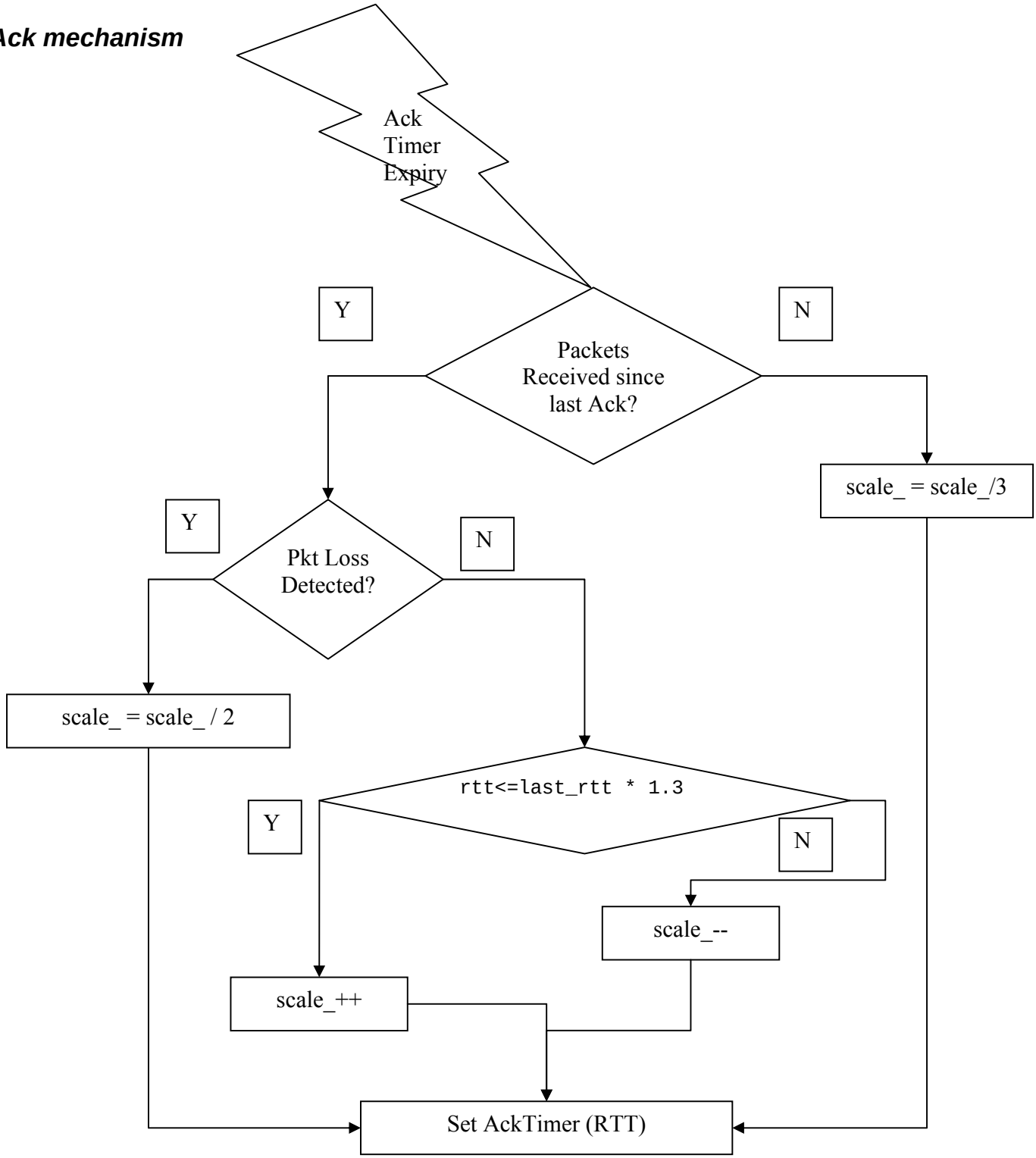
Sender Side



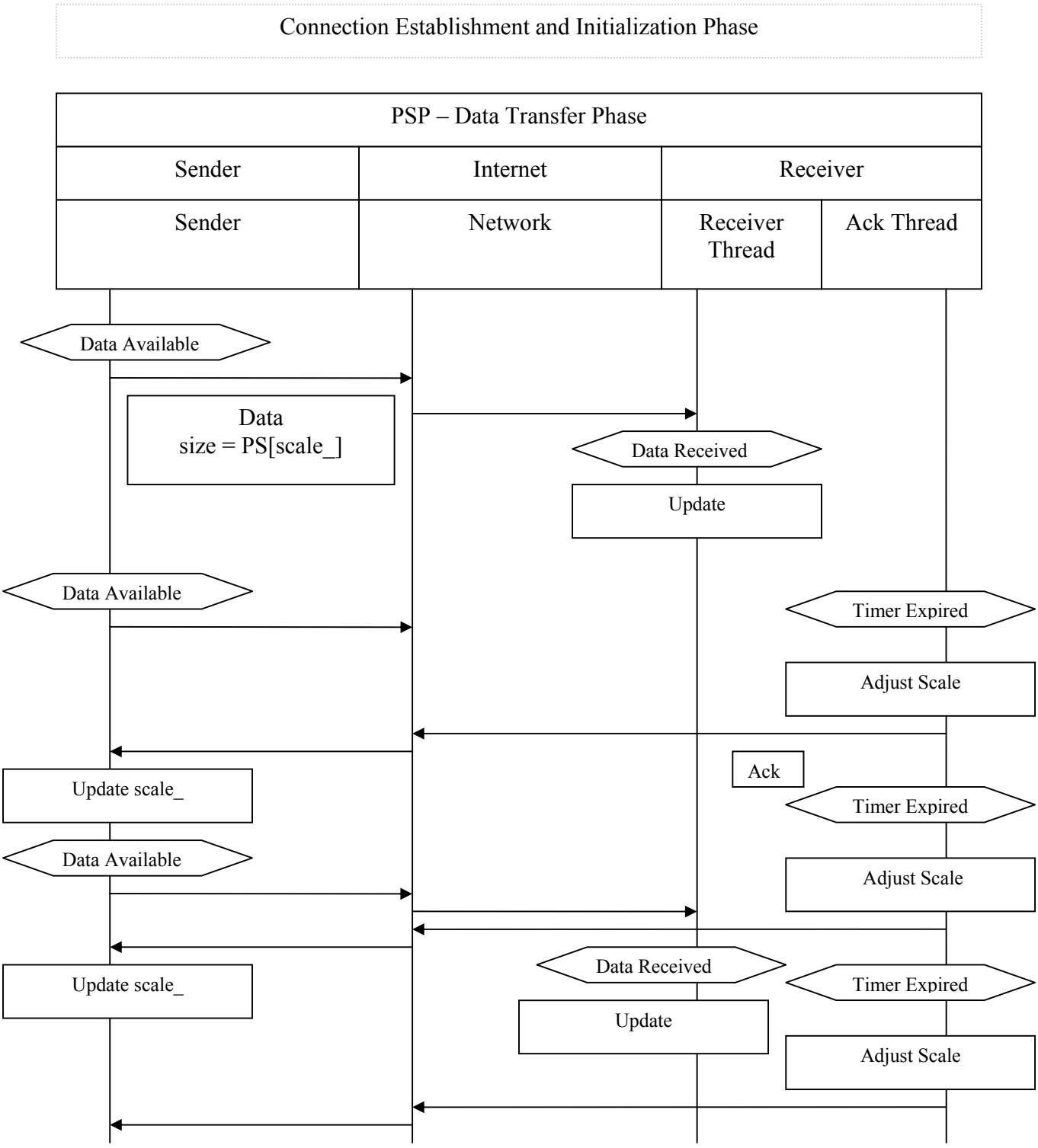
Receiver Side



Ack mechanism

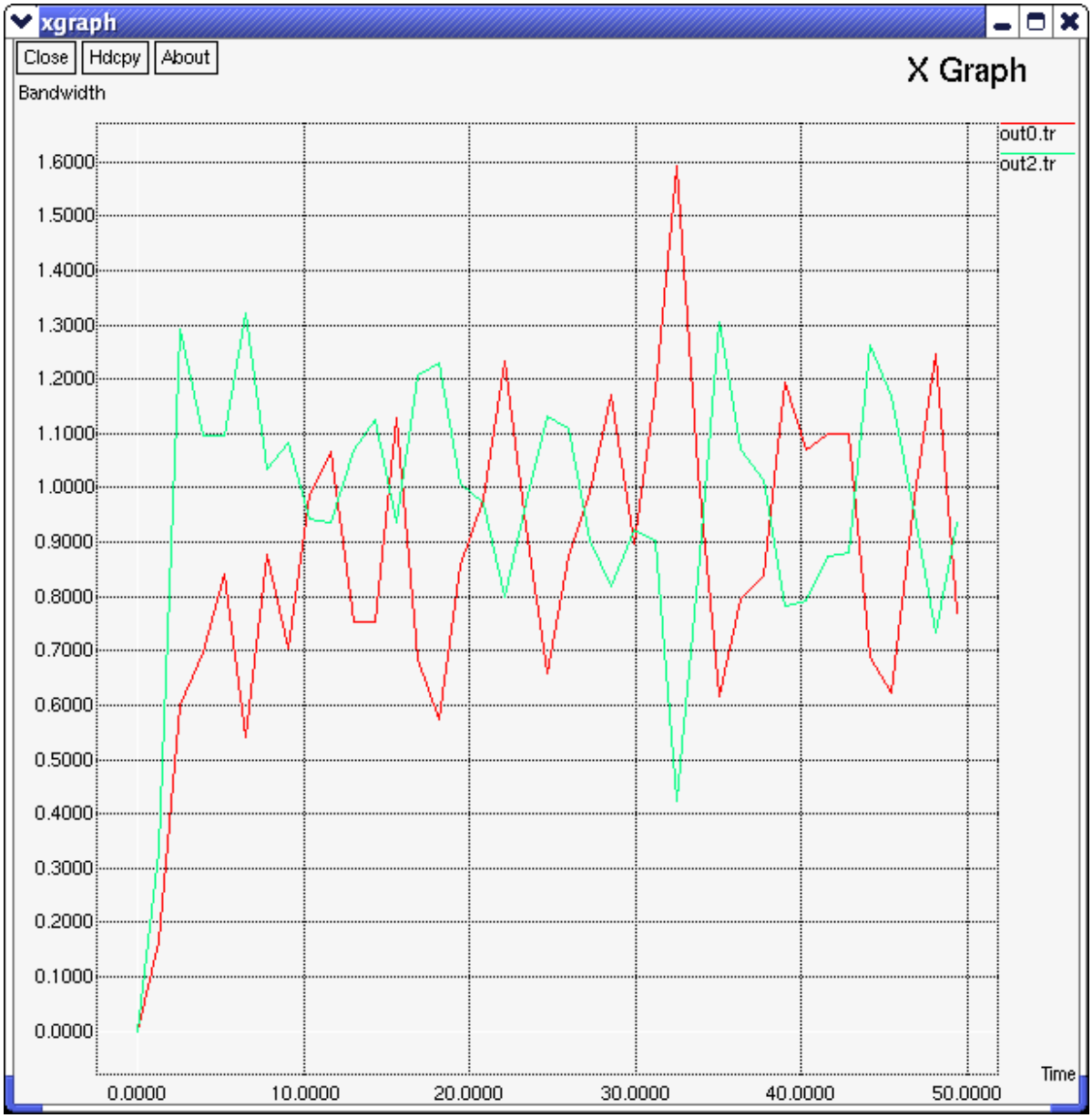


PSP Protocol Sequence Diagram



Results

TCP Vs. PSP (Throughput of the Sender)

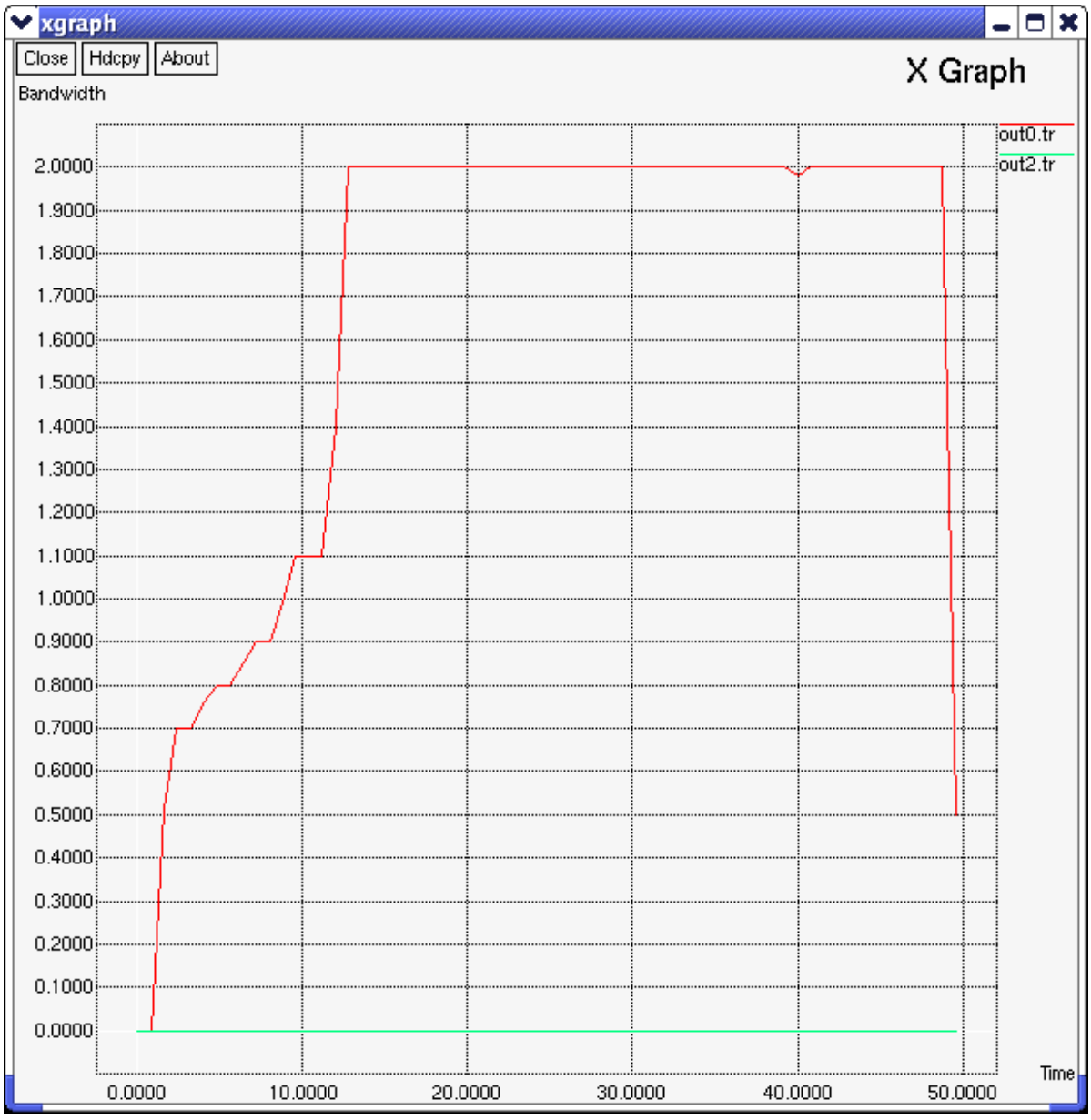


Units:
Bandwidth- mb/s
Time- Seconds

Legend:
PSP
TCP

This graph lets us conclude that PSP is in fact TCP-Friendly, because the average throughput of the PSP sender is less than or equal to the average throughput of the TCP sender. Also, there is a full utilization of the available resources, because at any time the sum of the throughputs of the competing flows equals the bottleneck bandwidth.

PSP Flow (after killing the TCP-flow)

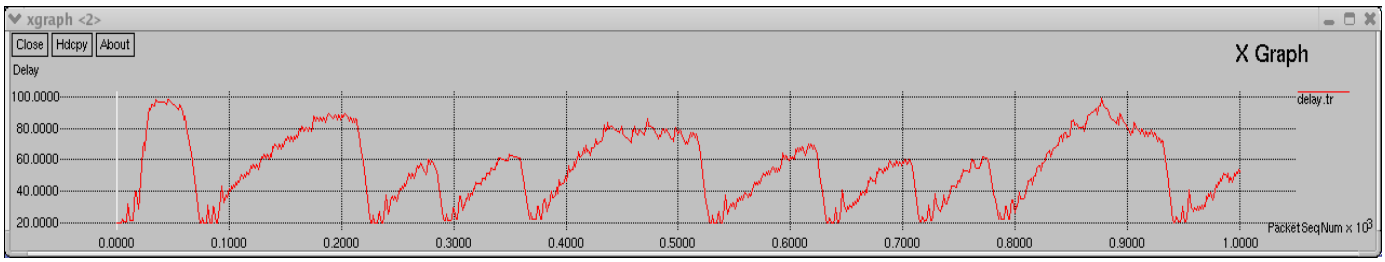


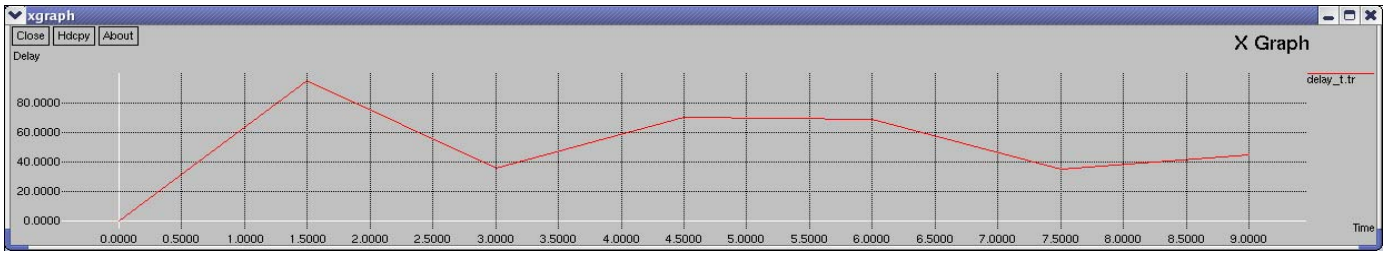
Units:
Bandwidth-mb/s
Time-Seconds

Legend:
PSP
TCP

This graph lets us conclude that the PSP-flow quickly achieves its fair share of the available resources.

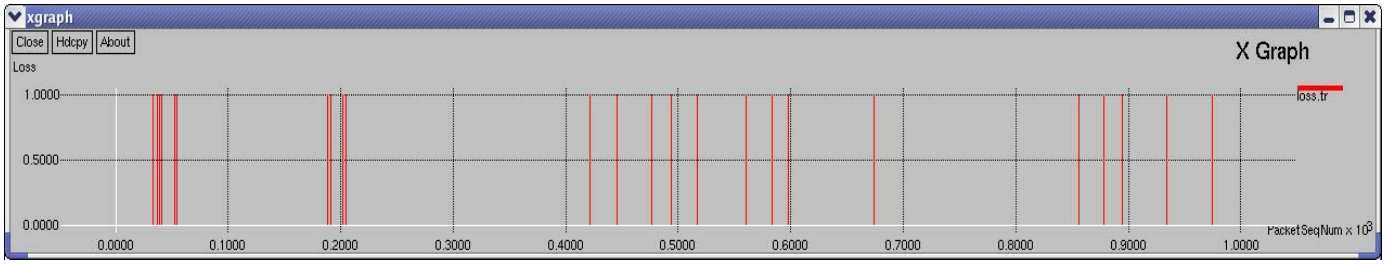
Delay Variation of PSP flow (co-existing TCP-flow)

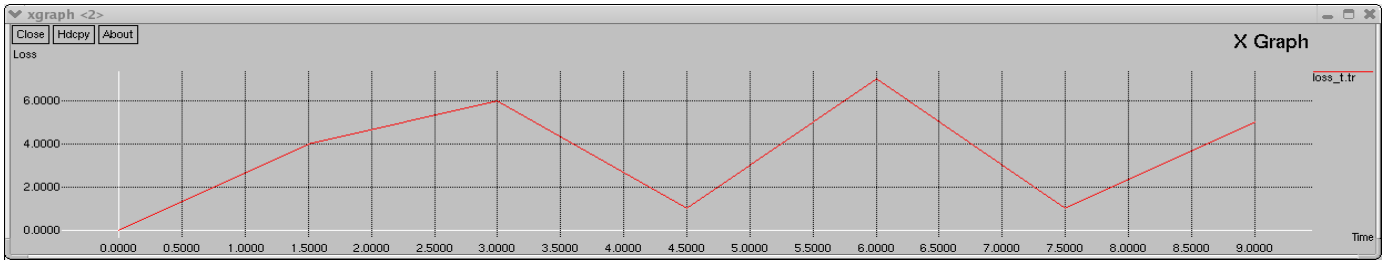




The first graph plots “End-to-End Packet Delay” Vs. “Packet Sequence Number” for the first 1000 packets. The second graph plots “End-to-End Packet Delay” Vs. “Time” for 9 seconds (roughly the time in which 1000 packets are sent). Sampling interval is 1.5 seconds.
These graphs let us conclude that the packet delay with PSP is suitable for VoIP-like applications, since these applications require an end-to-end delay of less than or equal to 150 ms.
The delay jitter also seems to be acceptable (the second graph).

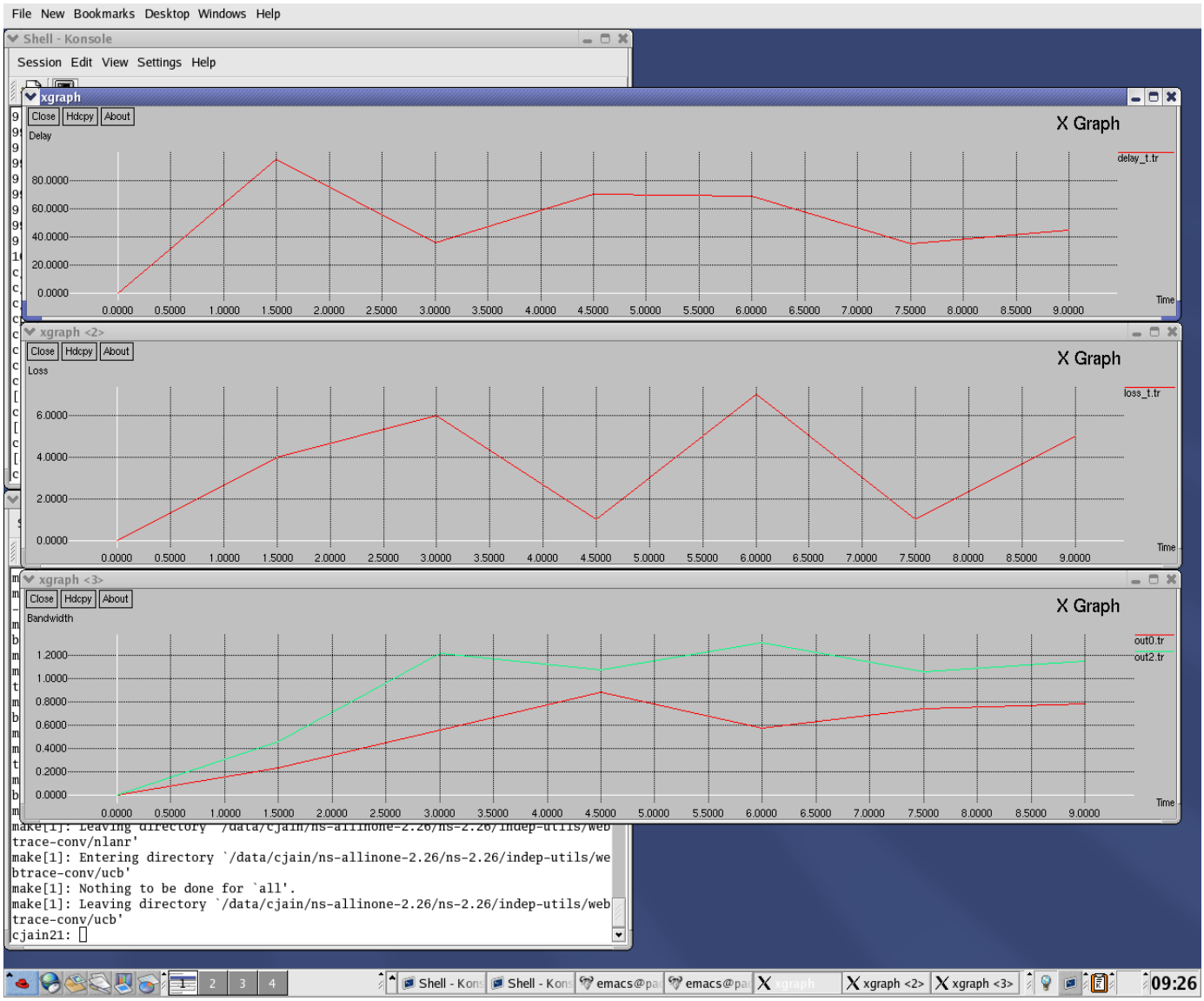
Packet Loss of PSP-flow (co-existing TCP-flow)





The first graph plots a bar graph for “Packet Loss” Vs. “Packet Sequence Number” for the first 1000 packets. A ‘1’ on the y-axis indicates that the corresponding packet was lost.
The second graph plots “Packet Loss” Vs. “Time” for 9 seconds (roughly the time in which 1000 packets are sent). Sampling interval is 1.5 seconds.
A 1% packet loss in the PSP-flow due to congestion was detected when it was operated along with a TCP-flow. Such a loss is acceptable for Internet Telephony applications.

At a Glance



These graphs depict

- "End-to-End Packet Delay" Vs "Time"
- "Packet Loss" Vs "Time"
- "Bandwidth" Vs "Time"

All the graphs are for a PSP-flow that operates alongside a TCP-flow. The monitoring time is 9 seconds and the sampling interval is 1.5 seconds.

A look at these graphs let us conclude that PSP does respond constructively to congestion. PSP perceives congestion through packet loss. As is clear from the second and the third graphs, as the packet loss increases, the transmission rate is reduced and as the congestion is cleared, the transmission rate increases. The delay is another mechanism that I have used to perceive

impending congestion and vary the rate likewise. However, this association is slightly depicted by the first and the third graphs. Please refer to “Detailed Flow-charts” for an explanation of the association between “Packet Delay” and “Transmission Rate”.

Discussion and Conclusions

- PSP is TCP-Friendly
- It quickly achieves its fair share of Bandwidth.
- No demands of high processing capacity at the receiver-end
- It is suitable for applications that choose to maintain a high rate at the expense of reduced packet size.
- Optimum results (for particular a simulation topology) were obtained when
    - PS_MAX ≤ MTU
    - Constant Rate, X ≤ (Bottleneck Bandwidth / PS_MAX)
    - PS_MIN ≥ PS_MAX / 4
- Packet loss owing to network congestion is between 1-2%, which is good as far as VoIP applications are concerned.

Future Work

- A connection establishment phase, in order to automatically populate the “Preset packet-sizes” table and settle upon an optimum constant transmission rate. These are important parameters in a Bandwidth limited environment to ensure a fair share of resources.
- Study PSP behavior with RED-PD.
- The receiver side is not immune to packet re-ordering and construes that as packet loss. One way to rectify this might be to employ TCP-like mechanism. We declare a packet as lost if three packets with higher sequence numbers have already arrived.
- Add code to handle Ack lost scenario.
- What compromises does a simplified mechanism like PSP entail? What applications can do away with it?

References

[1] S. Floyd and K. Fall, “Promoting the Use of End-to-end Congestion Control in the Internet,” *IEEE/ACM Trans. Net.*, vol. 7, no. 4, Aug. 1999, pp. 458–72.
[2] *Widmer, J.; Denda, R.; Mauve, M.;* “ A survey on TCP-friendly congestion control “ Network, IEEE , Volume:15 Issue:3, May-June 2001 Page(s): 28 -37
[3] Ramakrishnan, K., Floyd, S. and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.
[4] Balakrishnan, H., Rahul, H., and S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts," Proc. ACM SIGCOMM, Cambridge, MA, September 1999.
[5] Handley, M.; Floyd, S.; Padhye, J.; Widmer, J.; “TCP Friendly Protocol Specification (TFRC): Protocol Specification”; RFC 3448, January 2003.
[6] NS by Example http://nile.wpi.edu/NS/

Appendix

Code Listing

Packet-Size Scaling Protocol (Header File)

```
// File:      psp.h
// Author:    Charu Jain
// Modifed:   Nov., 29, 03

#include "timer-handler.h"
#include "packet.h"
#include "app.h"
#include "pudp.h"

const int MAXScale = 16; //16-level packet size scaling

// This is used for receiver's received packet accounting
struct pkt_accounting {
        int last_seq;   // sequence number of last received MM pkt
        int last_scale; // rate (0-4) of last acked
        int lost_pkts;  // number of lost pkts since last ack
        int recv_pkts;  // number of received pkts since last ack
        double rtt;     // round trip time
        double l_rtt;   //last rtt
};


class PSP;

// Sender uses this timer to
// schedule next app data packet transmission time
class SendTimer : public TimerHandler {
 public:
      SendTimer(PSP* t) : TimerHandler(), t_(t) {}
      inline virtual void expire(Event*);
 protected:
      PSP* t_;
};


// Reciver uses this timer to schedule
// next ack packet transmission time
class AckTimer : public TimerHandler {
 public:
      AckTimer(PSP* t) : TimerHandler(), t_(t) {}
      inline virtual void expire(Event*);
 protected:
      PSP* t_;
};
```

```
// Mulitmedia Application Class Definition
class PSP : public Application {
 public:
      PSP();
      void send_mm_pkt();  // called by SendTimer:expire (Sender)
      void send_ack_pkt(); // called by AckTimer:expire (Receiver)
 protected:
      int command(int argc, const char*const* argv);
      void start();       // Start sending data packets (Sender)
      void stop();        // Stop sending data packets (Sender)
 private:
      void init();
      inline double next_snd_time();                          // (Sender)
      virtual void recv_msg(int nbytes, const char *msg = 0); //
(Sender/Receiver)
      void set_scale(const hdr_mm *mh_buf);                   // (Sender)
      void adjust_scale(void);                                // (Receiver)
      void account_recv_pkt(const hdr_mm *mh_buf);            // (Receiver)
      void init_recv_pkt_accounting();                        // (Receiver)

      double ps[MAXScale];   // packet sizes associated to scale values
      double interval_;      // Application data packet transmission interval
      int pktsize_;          // Application data packet size
      int random_;           // If 1 add randomness to the interval
      int running_;          // If 1 application is running
      int seq_;              // Application data packet sequence number
      int scale_;            // Media scale parameter
      pkt_accounting p_accnt;
      SendTimer snd_timer_;  // SendTimer
      AckTimer  ack_timer_;  // AckTimer
      int sntBytes_;
      int recvBytes_;
      int pktsDrop_;
      double cra_;
};
```

Packet-Size Scaling Protocol (Implementation of Sender and Receiver, C++ file)

```
//
// File:      psp.cc
// Author:    Charu Jain
// Modifed:   Nov., 29, 03
//

#include "random.h"
#include "psp.h"


// PSP OTcl linkage class
static class PSPClass : public TclClass {
 public:
  PSPClass() : TclClass("Application/PSP") {}
  TclObject* create(int, const char*const*) {
    return (new PSP);
  }
} class_app_mm;
```

```
// When snd_timer_ expires call PSP:send_mm_pkt()
void SendTimer::expire(Event*)
{
  t_->send_mm_pkt();
}


// When ack_timer_ expires call PSP:send_ack_pkt()
void AckTimer::expire(Event*)
{
  t_->send_ack_pkt();
}


// Constructor (also initialize instances of timers)
PSP::PSP() : running_(0), snd_timer_(this), ack_timer_(this)
{
  bind_bw("ps0_", &ps[0]);
  bind_bw("ps1_", &ps[1]);
  bind_bw("ps2_", &ps[2]);
  bind_bw("ps3_", &ps[3]);
  bind_bw("ps4_", &ps[4]);
  bind_bw("ps5_", &ps[5]);
  bind_bw("ps6_", &ps[6]);
  bind_bw("ps7_", &ps[7]);
  bind_bw("ps8_", &ps[8]);
  bind_bw("ps9_", &ps[9]);
  bind_bw("ps10_", &ps[10]);
  bind_bw("ps11_", &ps[11]);
  bind_bw("ps12_", &ps[12]);
  bind_bw("ps13_", &ps[13]);
  bind_bw("ps14_", &ps[14]);
  bind_bw("ps15_", &ps[15]);
  bind("pktsize_", &pktsize_);
  bind("sntBytes_", &sntBytes_);
  bind("recvBytes_", &recvBytes_);
  bind("pktsDrop_", &pktsDrop_);
  bind("cra_", &cra_);
  bind_bool("random_", &random_);
}


// OTcl command interpreter
int PSP::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
      tcl.resultf("no such agent %s", argv[2]);
      return(TCL_ERROR);
      }
```

```
      // Make sure the underlying agent supports PSP Multimedia
      if(agent_->supportMM()) {
      agent_->enableMM();
      }
      else {
      tcl.resultf("agent \"%s\" does not support PSP Multimedia Application",
argv[2]);
      return(TCL_ERROR);
      }

      agent_->attachApp(this);
      return(TCL_OK);
    }
  }
  return (Application::command(argc, argv));
}



void PSP::init()
{
  scale_ = 0; // Start at minimum ps
  seq_ = 0;   // Sequence number (start from 0)
  interval_ = (double)(pktsize_ << 3)/cra_;
}


void PSP::start()
{
  init();
  running_ = 1;
  send_mm_pkt();
}


void PSP::stop()
{
  running_ = 0;
}


// Send application data packet
void PSP::send_mm_pkt()
{
  hdr_mm mh_buf;

  if (running_) {
    // the below info is passed to PUDP agent, which will write it
    // to MM header after packet creation.
    sntBytes_ += (int)ps[scale_];
    mh_buf.ack = 0;            // This is a MM packet
    mh_buf.seq = seq_++;         // MM sequece number
    mh_buf.nbytes = (int)ps[scale_];  // Size of MM packet
    //(NOT UDP packet size)
    mh_buf.time = Scheduler::instance().clock(); // Current time
    mh_buf.scale = scale_;                       // Current scale value
    agent_->sendmsg((int)ps[scale_], (char*) &mh_buf);  // send to UDP
```

```
    // Reschedule the send_pkt timer
    double next_time_ = next_snd_time();
    if(next_time_ > 0) snd_timer_.resched(next_time_);
  }
}


// Schedule next data packet transmission time
double PSP::next_snd_time()
{
  // Recompute interval in case rate or size chages
  interval_ = (double)(pktsize_ << 3)/cra_;
  double next_time_ = interval_;
  if(random_)
    next_time_ += interval_ * Random::uniform(-0.5, 0.5);
  return next_time_;
}


// Receive message from underlying agent
void PSP::recv_msg(int nbytes, const char *msg)
{
  if(msg) {
    hdr_mm* mh_buf = (hdr_mm*) msg;

    if(mh_buf->ack == 1) {
      // If received packet is ACK packet
      set_scale(mh_buf);
    }
    else {
      // If received packet is MM packet
      recvBytes_ += mh_buf->nbytes;
      account_recv_pkt(mh_buf);

      if(mh_buf->seq == 0) send_ack_pkt();
    }
  }
}


// Sender sets its scale to what reciver notifies
void PSP::set_scale(const hdr_mm *mh_buf)
{
  scale_ = mh_buf->scale;
}


void PSP::account_recv_pkt(const hdr_mm *mh_buf)
{
  double local_time = Scheduler::instance().clock();

  p_accnt.l_rtt =  p_accnt.rtt;

  // Calculate RTT
  if(mh_buf->seq == 0) {
    init_recv_pkt_accounting();
```

```
    p_accnt.rtt = 2*(local_time - mh_buf->time);
  }
  else
    p_accnt.rtt = 0.9 * p_accnt.rtt + 0.1 * 2*(local_time - mh_buf->time);


  // Count Received packets and Calculate Packet Loss
  p_accnt.recv_pkts ++;
  p_accnt.lost_pkts += (mh_buf->seq - p_accnt.last_seq - 1);
  if ((mh_buf->seq - p_accnt.last_seq - 1)>0)
    //printf("%f\t%d\n",local_time,1);
    pktsDrop_++;
  p_accnt.last_seq = mh_buf->seq;
}


void PSP::init_recv_pkt_accounting()
{
  p_accnt.last_seq = -1;
  p_accnt.last_scale = 0;
  p_accnt.lost_pkts = 0;
  p_accnt.recv_pkts = 0;
}


void PSP::send_ack_pkt(void)
{
  double local_time = Scheduler::instance().clock();

  adjust_scale();

  // send ack message
  hdr_mm ack_buf;
  ack_buf.ack = 1;  // this packet is ack packet
  ack_buf.time = local_time;
  ack_buf.nbytes = 40;  // Ack packet size is 40 Bytes
  ack_buf.scale = p_accnt.last_scale;
  agent_->sendmsg(ack_buf.nbytes, (char*) &ack_buf);

  // schedule next ACK time
  ack_timer_.resched(p_accnt.rtt);
}


void PSP::adjust_scale(void)
{
  if(p_accnt.recv_pkts > 0) {
    if(p_accnt.lost_pkts > 0)
      p_accnt.last_scale = (int)(p_accnt.last_scale / 2);
    else {
      if (p_accnt.l_rtt == 0) p_accnt.l_rtt = p_accnt.rtt;

      if(p_accnt.rtt <= (p_accnt.l_rtt * 1.0 ))
      {
        p_accnt.last_scale++;
        if(p_accnt.last_scale > MAXScale-1) p_accnt.last_scale = MAXScale-1;
```

```
      }

      if (p_accnt.rtt > (p_accnt.l_rtt * 1.19 ))//hmm..pkts are delayed more
      { //impending congestion operation undertaken!
        p_accnt.last_scale--;
        if(p_accnt.last_scale < 0) p_accnt.last_scale = 0;
      }
    }

    }

  else //ack timer expired before a packet arrived..must scale down drastically
       p_accnt.last_scale = (int)(p_accnt.last_scale / 3);

  p_accnt.lost_pkts = 0;
  //printf("%d\n%d\t%d\n", p_accnt.last_scale,p_accnt.recv_pkts, pktsDrop_);
}
```

Modified UDP-Agent (Header File)

```
//
// Author:    Charu Jain
// File:      pudp.h
// Modifed:   Nov., 29, 03
//

#ifndef ns_udp_mm_h
#define ns_udp_mm_h

#include "udp.h"
#include "ip.h"

// Multimedia Header Structure
struct hdr_mm {
      int ack;     // is it ack packet?
      int seq;     // mm sequence number
```

```
	int nbytes;  // bytes for mm pkt
	double time; // current time
	int scale;   // scale (0-MAXScale) associated with packet sizes

	// Packet header access functions
	  static int offset_;
	  inline static int& offset() { return offset_; }
	  inline static hdr_mm* access(const Packet* p) {
		  return (hdr_mm*) p->access(offset_);
	  }
};


// Used for Re-assemble segmented (by UDP) MM packet
struct asm_mm {
	int seq;     // mm sequence number
	int rbytes;  // currently received bytes
	int tbytes;  // total bytes to receive for MM packet
};


// PUDPAgent Class definition
class PUDPAgent : public UdpAgent {
public:
	PUDPAgent();
	PUDPAgent(packet_t);
	virtual int supportMM() { return 1; }
	virtual void enableMM() { support_mm_ = 1; }
	virtual void sendmsg(int nbytes, const char *flags = 0);
	void recv(Packet*, Handler*);
protected:
	int support_mm_; // set to 1 if above is MmApp
private:
	asm_mm asm_info; // packet re-assembly information
};

#endif
```

Modified UDP-Agent (C++ File)

```
//
// Author:    Charu Jain
// File:      pudp.cc
// Modifed:   Nov., 29, 03
//

#include "pudp.h"
#include "rtp.h"
#include "random.h"
#include <string.h>


int hdr_mm::offset_;
```

```
// Mulitmedia Header Class
static class MultimediaHeaderClass : public PacketHeaderClass {
public:
      MultimediaHeaderClass() : PacketHeaderClass("PacketHeader/Multimedia",
                                        sizeof(hdr_mm)) {
            bind_offset(&hdr_mm::offset_);
      }
} class_mmhdr;


// PUDPAgent OTcl linkage class
static class PUDPAgentClass : public TclClass {
public:
      PUDPAgentClass() : TclClass("Agent/UDP/PUDP") {}
      TclObject* create(int, const char*const*) {
            return (new PUDPAgent());
      }
} class_udpmm_agent;


// Constructor (with no arg)
PUDPAgent::PUDPAgent() : UdpAgent()
{
      support_mm_ = 0;
      asm_info.seq = -1;
}

PUDPAgent::PUDPAgent(packet_t type) : UdpAgent(type)
{
      support_mm_ = 0;
      asm_info.seq = -1;
}


// Add Support of Multimedia Application to PUDPAgent::sendmsg
void PUDPAgent::sendmsg(int nbytes, const char* flags)
{
      Packet *p;
      int n, remain;


      if (size_) {
            n = (nbytes/size_ + (nbytes%size_ ? 1 : 0));
            remain = nbytes%size_;
      }
      else
            printf("Error: PUDP size = 0\n");

      if (nbytes == -1) {
            printf("Error:  sendmsg() for PUDP should not be -1\n");
            return;
      }
      double local_time =Scheduler::instance().clock();
      while (n-- > 0) {
            p = allocpkt();
            if(n==0 && remain>0) hdr_cmn::access(p)->size() = remain;
```

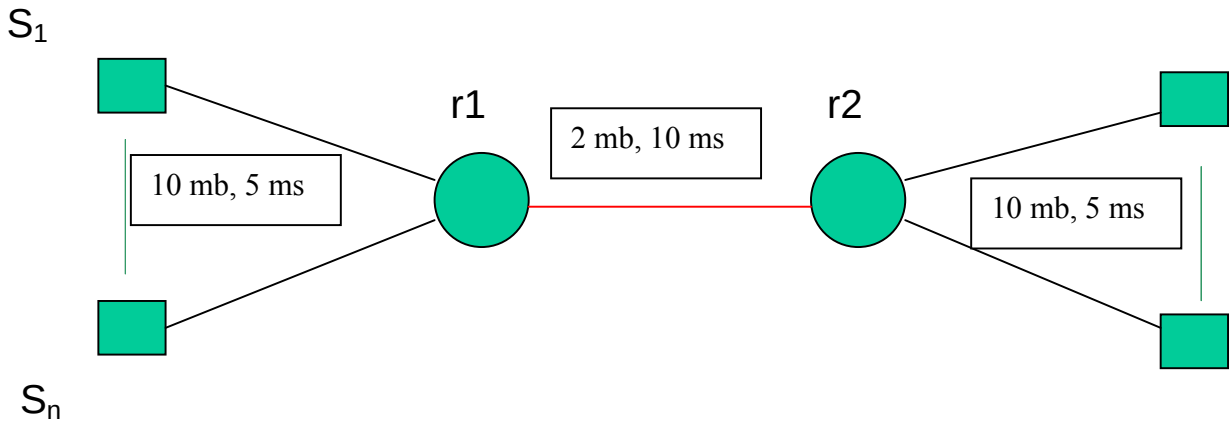
```
            else hdr_cmn::access(p)->size() = size_;
            hdr_rtp* rh = hdr_rtp::access(p);
            rh->flags() = 0;
            rh->seqno() = ++seqno_;
            hdr_cmn::access(p)->timestamp() =
                (u_int32_t)(SAMPLERATE*local_time);
            // to eliminate recv to use MM fields for non MM packets
            hdr_mm* mh = hdr_mm::access(p);
            mh->ack = 0;
            mh->seq = 0;
            mh->nbytes = 0;
            mh->time = 0;
            mh->scale = 0;
            // psp udp  packets are distinguished by setting the ip
            // priority bit to 15 (Max Priority).
            if(support_mm_) {
                  hdr_ip* ih = hdr_ip::access(p);
                  ih->prio_ = 15;
                  if(flags) // MM Seq Num is passed as flags
                        memcpy(mh, flags, sizeof(hdr_mm));
            }
            // add "beginning of talkspurt" labels (tcl/ex/test-rcvr.tcl)
            if (flags && (0 ==strcmp(flags, "NEW_BURST")))
                  rh->flags() |= RTP_M;
            target_->recv(p);
      }
      idle();
}


// Support Packet Re-Assembly and Multimedia Application
void PUDPAgent::recv(Packet* p, Handler*)
{
      hdr_ip* ih = hdr_ip::access(p);
      int bytes_to_deliver = hdr_cmn::access(p)->size();

      // if it is a MM packet (data or ack)
      if(ih->prio_ == 15) {
            if(app_) {  // if MM Application exists
                  // re-assemble MM Application packet if segmented
                  hdr_mm* mh = hdr_mm::access(p);
                  if(mh->seq == asm_info.seq)
                        asm_info.rbytes += hdr_cmn::access(p)->size();
                  else {
                        asm_info.seq = mh->seq;
                        asm_info.tbytes = mh->nbytes;
                        asm_info.rbytes = hdr_cmn::access(p)->size();
                  }
                  // if fully reassembled, pass the packet to application
                  if(asm_info.tbytes == asm_info.rbytes) {
                        hdr_mm mh_buf;
                        memcpy(&mh_buf, mh, sizeof(hdr_mm));
                        app_->recv_msg(mh_buf.nbytes, (char*) &mh_buf);
                  }
            }
            Packet::free(p);
      }
```

```
        // if it is a normal data packet (not MM data or ack packet)
        else {
                if (app_) app_->recv(bytes_to_deliver);
                Packet::free(p);
        }
}
```

Simulation Topology and TCL Simulation Script




```
# Author: Charu Jain

# Date:   Nov., 30, 2003


set ns [new Simulator]


#Define different colors for data flows

$ns color 1 Red

$ns color 2 Blue



#Open the nam and xgraph trace file

set nf [open out.nam w]

set tf [open out.tr w]

set f0 [open out0.tr w]
```

```
set f1 [open out1.tr w]

set f2 [open out2.tr w]

set f3 [open out3.tr w]

set f4 [open out4.tr w]

set f5 [open out5.tr w]


set temp 0


$ns namtrace-all $nf

$ns trace-all $tf


#Define a 'finish' procedure

proc finish {} {

global ns nf tf f0 f1 f2 f3 f4

$ns flush-trace

#Close the trace file

close $nf

close $tf

close $f0

close $f1

close $f2

close $f3

close $f4
```

```
#Execute nam and xgraph on the output file
exec xgraph -bb  out0.tr out2.tr -x Time -y Bandwidth &
exec nam out.nam &
exit 0
}

set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(r1) [$ns node]
set node_(r2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]

$ns duplex-link $node_(s1) $node_(r1) 5Mb 3ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 5Mb 3ms DropTail
$ns duplex-link $node_(r1) $node_(r2) 2Mb 10ms RED
$ns duplex-link $node_(s3) $node_(r2) 5Mb 3ms DropTail
$ns duplex-link $node_(s4) $node_(r2) 5Mb 3ms DropTail

#Setup RED queue parameter
$ns queue-limit $node_(r1) $node_(r2) 20
Queue/RED set thresh_ 5
Queue/RED set maxthresh_ 10
Queue/RED set q_weight_ 0.002
```

```
Queue/RED set ave_ 0

$ns duplex-link-op $node_(r1) $node_(r2) queuePos 0.5

$ns duplex-link-op $node_(s1) $node_(r1) orient right-down
$ns duplex-link-op $node_(s2) $node_(r1) orient right-up
$ns duplex-link-op $node_(r1) $node_(r2) orient right
$ns duplex-link-op $node_(s3) $node_(r2) orient left-down
$ns duplex-link-op $node_(s4) $node_(r2) orient left-up


#Setup a PUDP Agent
set udp_s [new Agent/UDP/PUDP]
set udp_r [new Agent/UDP/PUDP]
$ns attach-agent $node_(s1) $udp_s
$ns attach-agent $node_(s3) $udp_r
$ns connect $udp_s $udp_r
$udp_s set packetSize_ 2000
$udp_r set packetSize_ 2000
$udp_s set fid_ 1
$udp_r set fid_ 1

#Setup a PSP Application and connect with PUDP Agent
set pspapp_s [new Application/PSP]
```

```
set pspapp_r [new Application/PSP]

$pspapp_s attach-agent $udp_s

$pspapp_r attach-agent $udp_r

$pspapp_s set pktsize_ 1000

$pspapp_s set cra_ 1000000.0

$pspapp_s set random_ false

#Setup a TCP connection

set sprot0 [new Agent/TCP]

set sink0 [new Agent/TCPSink]

$ns attach-agent $node_(s2) $sprot0

$ns attach-agent $node_(s4) $sink0

$ns connect $sprot0 $sink0

$sprot0 set fid_ 2

#Setup a CBR traffic generator
set traf0 [new Application/Traffic/CBR]

$traf0 attach-agent $sprot0

proc record {} {

global pspapp_s pspapp_r sink0 sprot0 f0 f1 f2 f3 f4 temp

#Get an instance of the simulator

set ns [Simulator instance]

#Set the time after which the procedure should be called again
```

```
set time 1.5


#Count the number of bytes sent and recieved by pspapp

set bw0 [$pspapp_s set sntBytes_]

set bw1 [$pspapp_r set recvBytes_]

#for recvBytes of TCP

set bw3 [$sink0 set recvBytes_]


#for sent bytes of TCP

set bw2 [expr [$sprot0 set ndatabytes_] - $temp]

set temp [$sprot0 set ndatabytes_]


#Get the current time

set now [$ns now]


#Calculate the bandwidth (in MBit/s) and write it to the files

puts $f0 "$now [expr $bw0/$time*8/1000000]"

puts $f3 "$now [expr $bw3/$time*8/1000000]"
puts $f1 "$now [expr $bw1/$time*8/1000000]"

puts $f2 "$now [expr $bw2/$time*8/1000000]"


#Reset the bytes_ values on the traffic sinks and sources

$pspapp_s set sntBytes_ 0
```

```
$pspapp_r set recvBytes_ 0

$pspapp_r set pktsDrop_ 0

$sink0 set recvBytes_ 0


#Re-schedule the procedure

$ns at [expr $now+$time] "record"

}



#Simulation Scenario

$ns at 0.0 "record"

$ns at 1.0 "$traf0 start"

$ns at 1.0 "$pspapp_s start"


$ns at 9.0 "$traf0 stop"

$ns at 9.0 "$pspapp_s stop"


$ns at 10.0 "finish"


$ns run
```